OBJECT ORIENTED PROGRAMMING IN C++ [UNIT-III]

Learning Objectives

- Extending classes
- Types of Inheritance
- Defining a derived class
- Inheriting private members
- Virtual, Direct & Indirect base class
- Defining derived class constructors
- Overriding inheritance method
- Nesting of Classes

Inheritance

- Inheritance is the property of one class to inherit the properties of another class.
- Major reason behind inheritance is reusability.
- The mechanism of deriving a new class from an old or existing class is called **inheritance (derivation).**
- The old class is referred to as the **base class** and new class is called **derived class** or **sub class.** The derived class inherits some or all of the traits from the base class.
- A class can also inherit properties from more than one class, for more than one level.

Inheritance Examples

Base Class	Derived Classes
Shape	Circle Triangle Rectangle
Loan	CarLoan HomeImprovementLoan MortgageLoan

Derived class declaration

- Specifies its relationship with the base class in addition to its own features

```
Class DerivedClass:[ VisibilityMode] BaseClass
{
    //member of derived class
    //can access member of base class
}
```

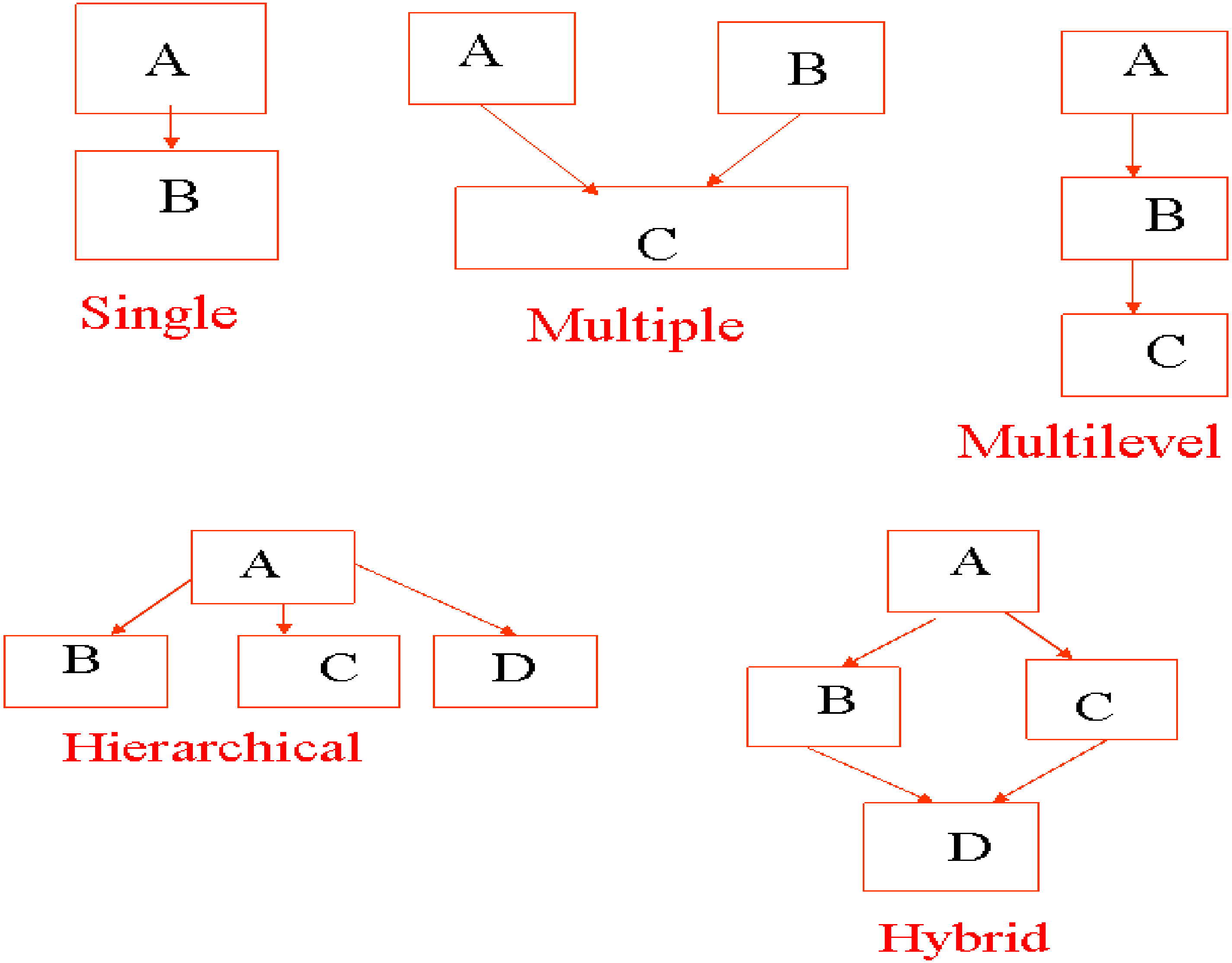
visibility mode

- Three types of visibility mode
  - public
  - protected
  - private
- Default visibility mode is private

Forms of Inheritance

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

Types of Inheritance



Deriving a Class

- The private derivation means that the derived class can access the public and the protected members of the base class privately.
- With privately derived class, the **public and the protected members of the base class become private members of the derived class.**

```
derived_class_name: acess_specifier base_class_name(<argument_list>)
```

Visibility Of Inherited Members

Base class visibility	**Derived class visibility**		
	Public	**Private**	**Protected**
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Visibility Of Inherited Members

Example -objects of type **derived can directly access the** public members of **base:**

```
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b)
{ i=a; j=b;
}
void show() {
cout << i << " " << j << "\n"; }
};
class derived : public base
{int k;
public:
derived(int x)
{ k=x; }
void showk() { cout << k << "\n"; }};
int main()
{derived ob(3);
ob.set(1, 2); // access member of base
ob.show(); // access member of base
ob.showk(); // member of derived class
return 0;
}
```

Visibility Of Inherited Members

When the base class is inherited by using the **private access specifier, all public and protected members of the base class become private members of the derived class.** For example, the following program will not even compile because **both set( ) and show( ) are now private elements of derived:**

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n";}
};
```

Visibility Of Inherited Members

```
// Public elements of base are private in derived.
class derived : private base
{
int k;
public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob(3);
ob.set(1, 2); // error, can't access set()
ob.show(); // error, can't access show()
return 0;
}
```

Visibility Of Inherited Members

Protected members behave differently. If the base class is inherited a **public, then** the base class' protected members become protecte members of the derived class and are, therefore, accessible by th derived class. By using **protected, you can create class** members tha are private to their class but that can still be inherited and accessed by derived class.

```
#include <iostream>
using namespace std;
class base {
protected:
int i, j; // private to base, but accessible by derived
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
```

Visibility Of Inherited Members

```
class derived : public base {
int k;
public:
// derived may access base's i and j
void setk() { k=i*j; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob;
ob.set(2, 3); // OK, known to derived
ob.show(); // OK, known to derived
ob.setk();
ob.showk();
return 0;
}
```

Visibility Of Inherited Members

In this example, because **base is inherited by derived as public and because i and j are declared as protected, derived's function setk( ) may access them. If i and j had** been declared as **private by base, then derived would not have access to them, and the** program would not compile.

Visibility Of Inherited Members

It is possible to inherit a base class as **protected. When this is done, all public and** protected members of the base class become protected members of the derived class.

For example,

```
#include <iostream>
using namespace std;
class base {
protected:
int i, j; // private to base, but accessible by derived
public:
void setij(int a, int b) { i=a; j=b; }
void showij() { cout << i << " " << j << "\n"; }
};
```

Visibility Of Inherited Members

```
// Inherit base as protected.
class derived : protected base
{
int k;
public:
// derived may access base's i and j and setij().
void setk()
{
setij(10, 12);
k = i*j;
}
// may access showij() here
void showall()
 {
cout << k << " ";
showij();
}
};
```

Visibility Of Inherited Members

```
int main()
{
derived ob;
// ob.setij(2, 3); // illegal, setij() is  protected member of derived
ob.setk(); // OK, public member of derived
ob.showall(); // OK, public member of derived
// ob.showij(); // illegal, showij() is protected  member of derived
return 0;
}
```

As you can see by reading the comments, even though **setij( ) and showij( ) are** public members of **base, they become protected members of derived when it is** inherited using the **protected access specifier. This means that they will not be** accessible inside **main( ).**

Visibility Of Inherited Members

It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived inherits both base1 and base2.**
An example of multiple base classes.

```
#include <iostream>
using namespace std;
class base1 {
protected:
int x;
public:
void showx() { cout << x << "\n"; }
};
class base2 {
protected:
int y;
public:
```

Visibility Of Inherited Members

```
void showy() {cout << y << "\n";}
};
// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
void set(int i, int j) { x=i; y=j; }
};
int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
ob.showy(); // from base2
return 0;
}
```

Visibility Of Inherited Members

how an access declaration works,Ex-

```
class base {
public:
int j; // public in base
};
// Inherit base as private.
class derived: private base {
public:
// here is access declaration
base::j; // make j public again
.
.
.
};
```

Because **base is inherited as private by derived, the public member j is made a private** member of **derived. However, by including**

base::j;

Program of Single Inheritance

```
#include<iostream.h> #include<conio.h>
class person //Base class or Super class
{ char name[20]; int age; public: void read_data(); void display_data();};
class student : public person // Derived class or Sub class
{ int roll; int marks; char grade;
public : void get_data(); char compute_grade(); void show_data(); };
void person :: read_data()
{ cout<<”\n Enter name:”; cin>>name; cout<<”\n Enter age: “;
   cin>>age;}
void person :: display_data()
{cout<<”\n Name: “<< name; cout<< “\n Age: “ << age; }
```

Program of Single Inheritance

```
 void student :: get_data()
{read_data();   cout<< "\n Enter Roll :"; cin>>roll;
cout<< "\n Enter Marks: "; cin>> marks; grade = compute_grade();}
char student :: compute_grade()
{char gd; if (marks<80)    gd = 'B' else gd = 'A'; return (gd);}
void student :: show_data()
{cout << "\n Roll: " << roll;cout << "\n Marks: " << marks;
cout<< "\n Grade: " <<grade;}
 main(){student s1; // Create an object of student type
s1.get_data();// Read data of a student
cout<<"\n The student data is…:; cout<<"\n";
obj.dispaly_data(); obj.show_data(); return(0); }
```

Multilevel Inheritance

```
#include<iostream.h> #include<conio.h>
class A1{protected: char name[15]; int age; };
class A2:public A1 // First level derivation
{ protected: float height; float weight; };
class A3:public A2 // Second level derivation
protected: char sex; public: void get()  // Reads data
{ cout<<"Name: "; cin>> name; cout<<"Age: "; cin>> age;
cout<<"Sex: "; cin>> sex; cout<<"Height: "; cin>> height;
cout<<"Weight: "; cin>> weight; } void show() // Displays data
cout<<"\n Name: " << name; cout<<"\n Age: " << age<< "Years";
cout<<"\n Sex: " << sex; cout<<"Height: " << height << "Feets";
cout<<"Weight: " << weight << "kg";}};
```

Multilevel Inheritance

```
void main()
{
A3 X;          //  Object declaration
X.get();      // Reads data
X.show();  // Displays data
}
```

Multiple Inheritance

```
#include<iostream.h>
class A {protected: int a;};// class A declaration
class B {protected: int b;};// class B declaration
class C {protected: int c;};// class C declaration
class D {protected: int d;};// class D declaration
class E: public A, B, C, D  // Multiple Derivation
{int e;
public:
void getdata()
{cout<<"\n Enter values of a, b, c, d & e: ";
cin>>a>>b>>c>>d>>e;}
```

Multiple Inheritance

```
void showdata()
{cout<<"\n a= " <<a <<"b=  " <<b<< "c = " << c << "d = " <<d <<
    "e= " <<e;}};
 void main()
{E   x;                    // Object declaration
x.getdata();          // Reads data
x.showdata();     //Displays data
}
```

<u>OUTPUT:</u>

Enter values of a, b, c, d & e: 1 2 4 8 16

a=1 b=2 c=4 d=8 z=16

Example of Hierarchial Inheritance

Instead of starting from the scratch, we can simply derive a new class employee from the base class person.

#include<iostream.h> #include<conio.h>

class person //Base class or Super class

{char name[20]; int age;

public: void read_data();void display_data();};

class student : public person // Derived class or Sub class

{int roll; int marks; char grade; public : void get_data();

char compute_grade(); void show_data();};

class employee : public person

{float bp; float hr; float sal;

public:void input_emp(); float compute_salary(); void disp_emp();};

Example of Hierarchial Inheritance

```
void employee :: input_emp()
{read_data(); // Read name & age from the base class
cout<< "\n Enter Basic Pay:"; cin >> bp;
cout<<"\n Enter HRA :"; cin>>hr;
sal = compute_salary();}
 float employee :: compute_salary()
{float total; total=bp+hr+2.5*bp; return(total);}
void employee :: disp_emp()
{cout<< "\n B.P. : " << bp; cout<< "\n H.R. : " << hr;
cout<< "\n Salary : " << sal; }
```

Example of Hierarchial Inheritance

```
void person :: read_data()
{cout<<"\n Enter name:"; cin>>name;
cout<<"\n Enter age: "; cin>>age;}
 void person :: display_data()
{cout<<"\n Name: "<< name;
cout<< "\n Age: " << age;}
 void student :: get_data()
{read_data();   //read name & age from base class
                // Reusability of code from base class
cout<< "\n Enter Roll :"; cin>>roll;
cout<< "\n Enter Marks: "; cin>> marks;
grade = compute_grade();}
```

Example of Hierarchial Inheritance

```
char student :: compute_grade()
{char gd;
if (marks<80)
    gd = ’B’
else gd = ‘A’;
return (gd);}
void student :: show_data()
{
cout << “\n Roll: “ << roll;
cout << “\n Marks: “ << marks;
cout<< “\n Grade: “ <<grade;
}
```

Example of Hierarchial Inheritance

```
main()
{student s1; // Create an object of student type
employee e1; // Create an object of employee tppe
s1.getdata();// Read data of a student
cout<<"\n The student data is…:;
obj.dispaly_data();//inherit function from class person
obj.show_data();
e1.input_emp(); // Read Employee data
cout<<"\n The Employee data is…:;
e1.dispay_data ();// Inherit function from class person to display
e1. disp_emp ();   // Display Employee details
return(0);}
```

Base class members		How base class members appear in the derived class
private: X protected: Y public: Z	private base class →	X is inaccessible. private: Y private: Z
private: X protected: Y public: Z	protected base class →	X is inaccessible. protected: Y protected: Z
private: X protected: Y public: Z	public base class →	X is inaccessible. protected: Y public: Z

Over loaded Member Functions

- The members of the derived class can have the same name as those defined in the base class
- If the same member exist in both the base class and the derived class, the member in the derived class will be executed
- The member of the base class can be access using scope resolution with overriding functions
- The general form is

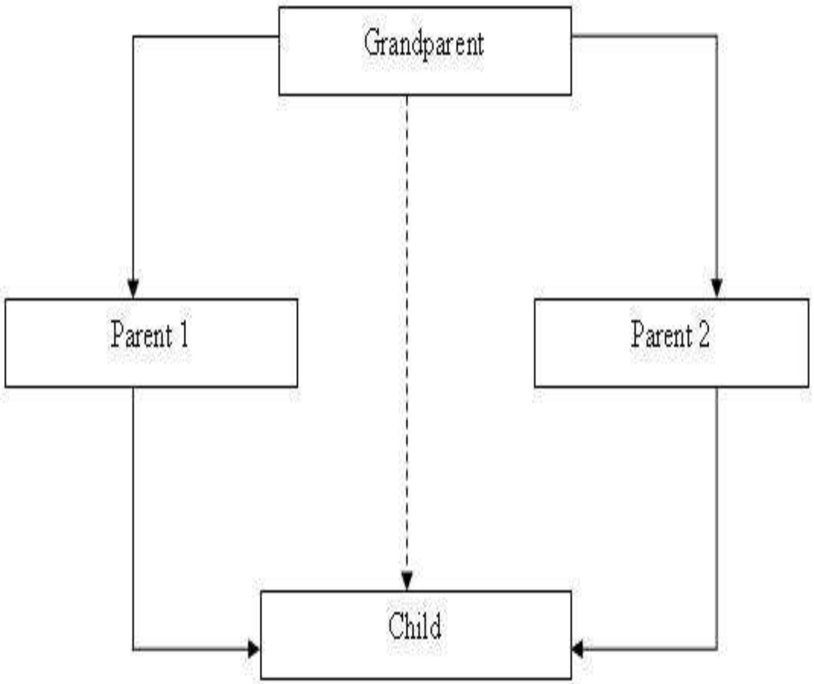
  Classname :: Membername ( ) ;

Over loaded Member Functions

```
class a{
        public :
                void fn(){ cout<<"base\n";}};
class b: public a{
public :
        void fn(){ cout<<"derived\n";}};
void main(){
        b obj;
        obj.fn();
        obj.a::fn();      }
```

Virtual Base Classes

- A situation may arise where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance are involved.
- This is illustrated in the figure where the child has two direct base classes 'parent1' and 'parent2' which themselves have a common base class 'grandparent'.
- The 'child' inherits the behaviors of grandparent class via two separate paths.
- It can also inherit directly, as shown by the dotted lines.
- The grandparent class is sometimes referred to as indirect base class.

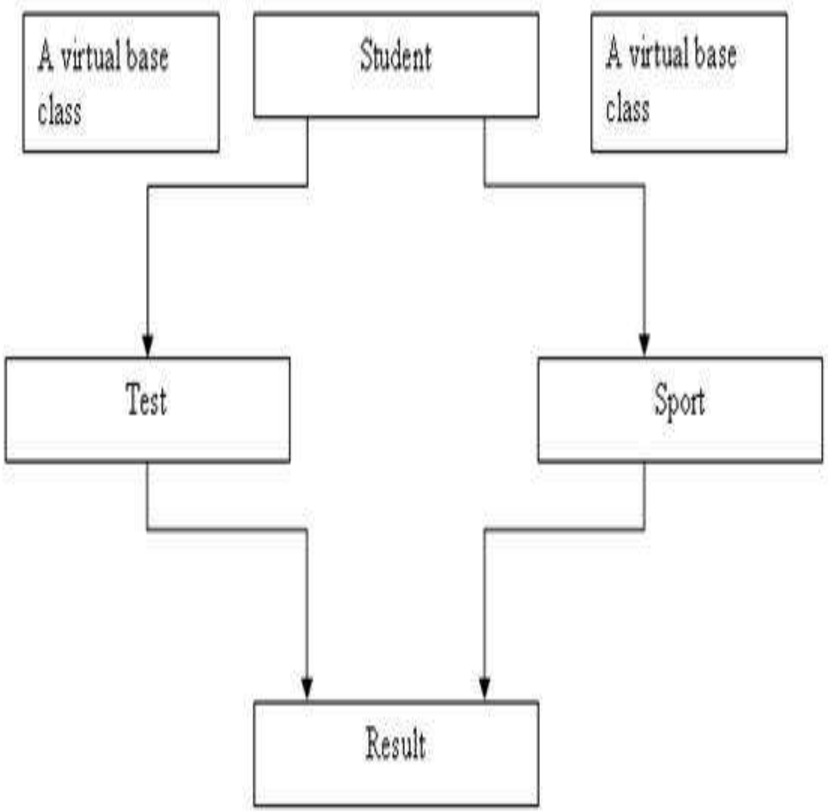


Virtual Base Classes

- Such inheritance by the child class may create some problems.
- **All the public and protected members of the grandparent class are inherited into the child class twice; first via parent1 class and then again via parent2 class.**
- **This means that the child class would have duplicate set of members inherited from grandparent, which introduces ambiguity and should be avoided.**
- **The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class,** while declaring the direct or intermediate base classes.

Virtual Base Classes

- **When a class is made virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.**
- Example: Student Result Processing System.



Associations

Employee

name: string
address: string
dateOfBirth: Date
employeeNo: integer
socialSecurityNo: string
department: Dept
manager: Employee
salary: integer
status: {current, left, retired}
taxCode: integer
. . .

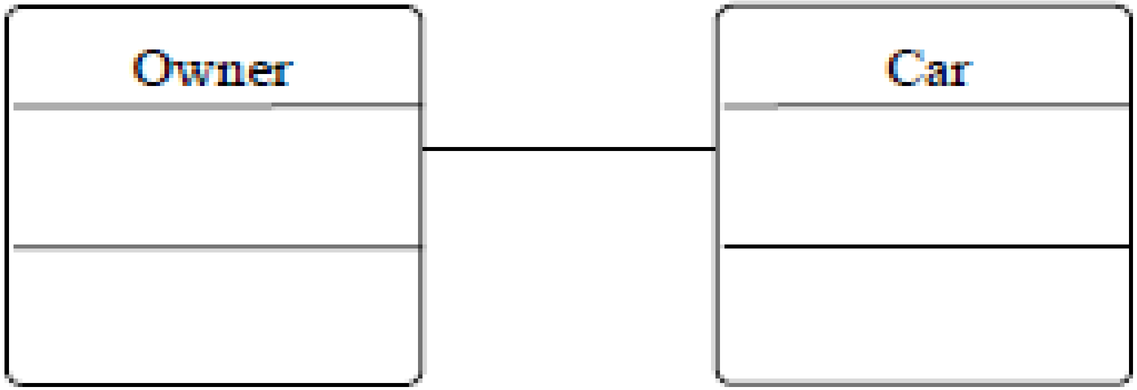
join ()
leave ()
retire ()
changeDetails ()

Associations

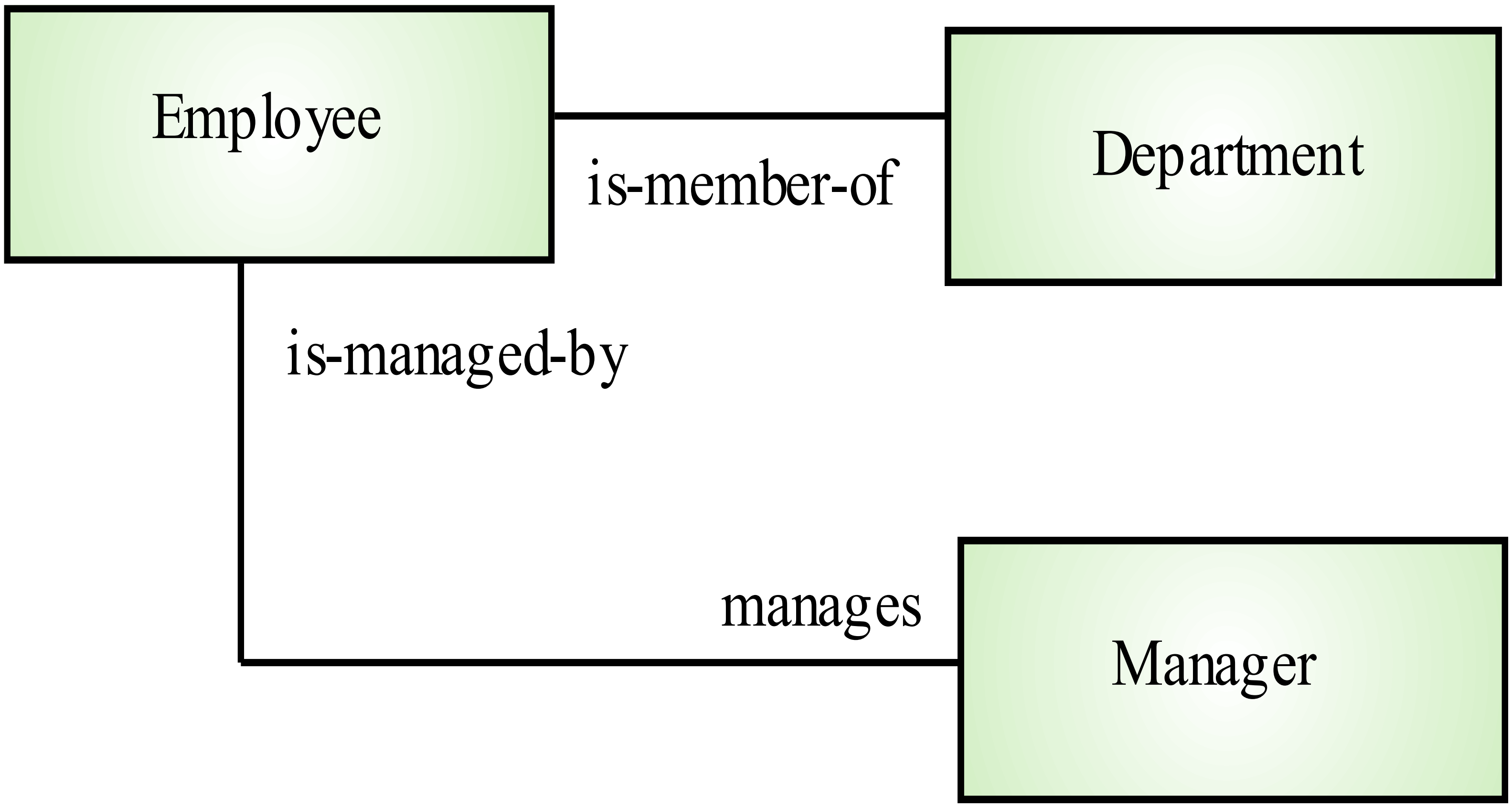
• A semantic **relationship between two or more classes** that specifies connections among their instances.

• Example: —**An Employee works for a Company**

An association can be viewed as a **weak form of aggregation** or as a data-oriented relationship between **two entities**. For example, **if there is a car, it must be associated with an owner.**



Associations



Generalization

- Objects are members of classes which define attribute types and operations
- Classes may be arranged in a class hierarchy where **one class (a super-class) is a generalisation of one or more other classes (sub-classes)**
- A sub-class **inherits the attributes** and operations from its super class and may add new methods or attributes of its own
- Generalisation in the UML is implemented as inheritance in OO programming languages

Generalization

- Inheritance is also called the generalization-specialization, gen-spec, or **IsA hierarchy.**
- Note that inheritance is a **relationship between classes**, not objects.
- **Generalized classes are placed higher in the hierarchy while specialized ones are found below.**
- For example, a **Vehicle is a generalized class** while **TruckVehicle and CarVehicle are more specialized ones.**
- In other words, a **TruckVehicle is-a-kind-of Vehicle.**

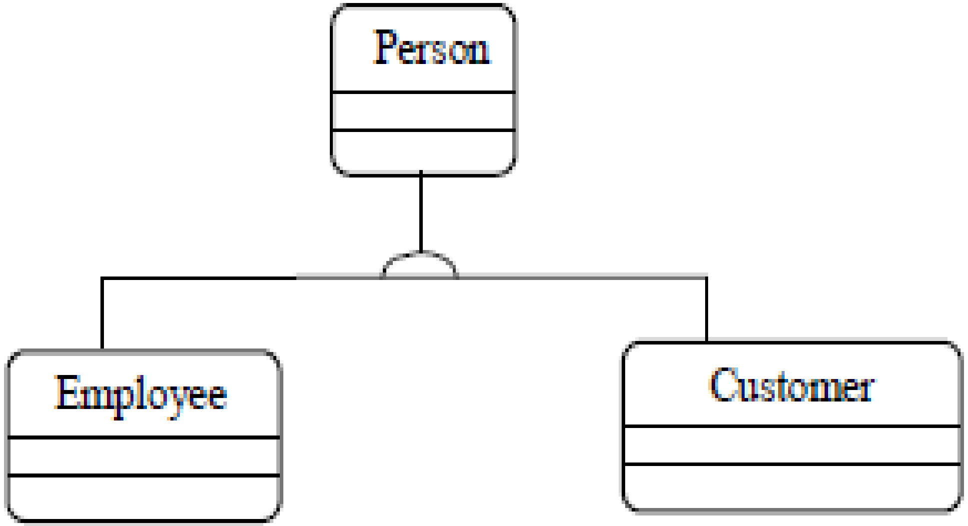
Generalization

Inheritance hierarchy.

• The **generalized class at the top** and the **specialized classes below.**

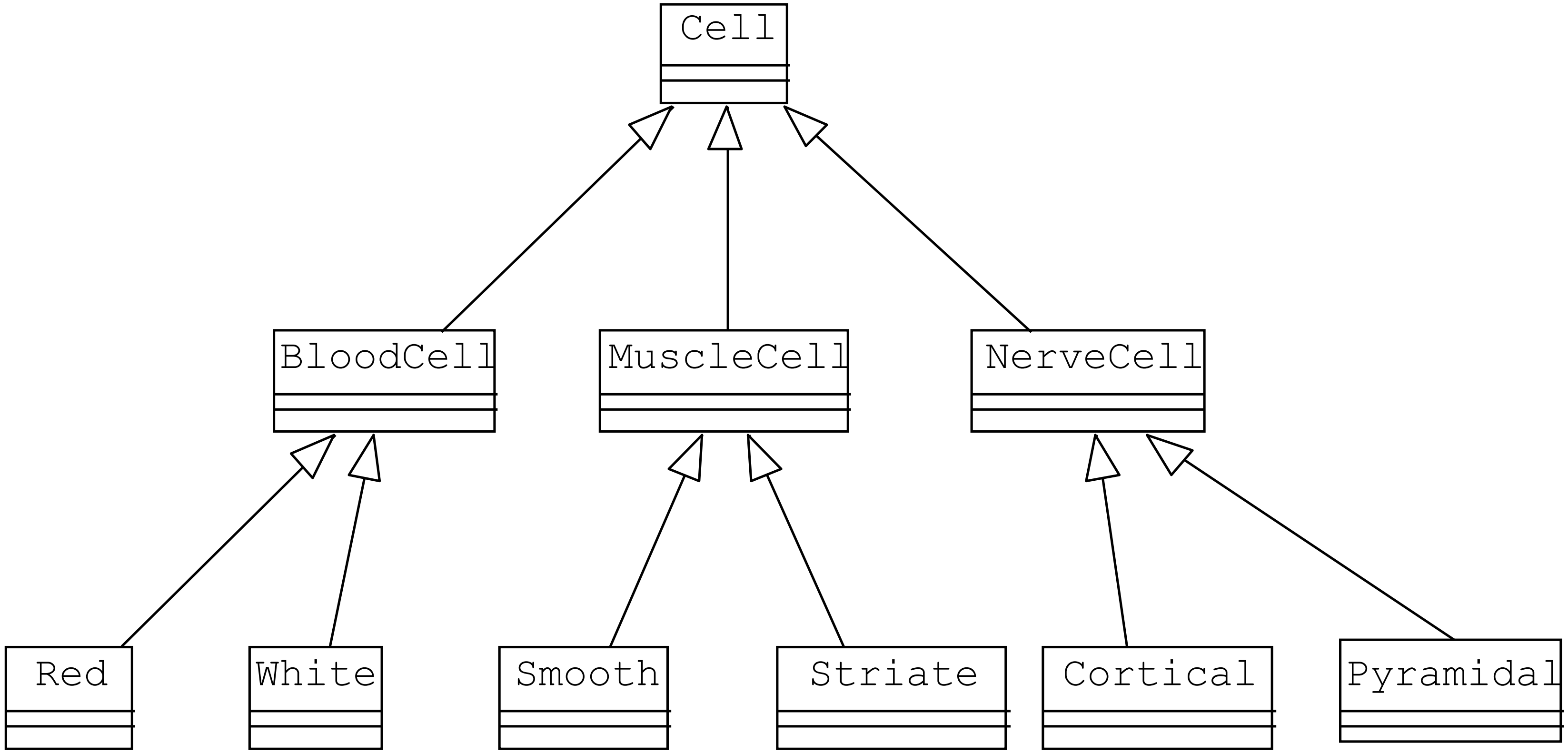
• The specialized class names should reflect the class they were specialized from.

• For example, employee was specialized from Person.

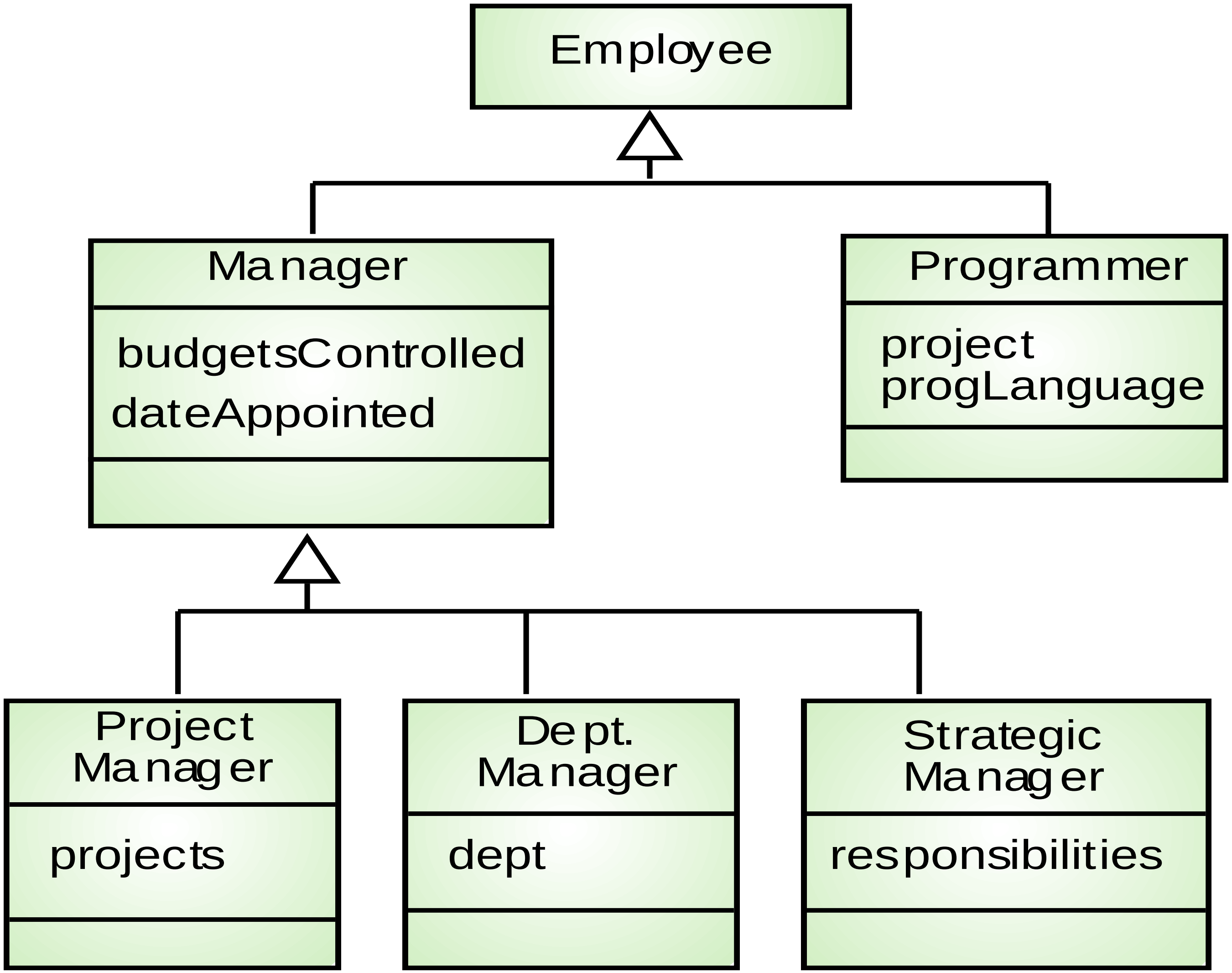


Generalization

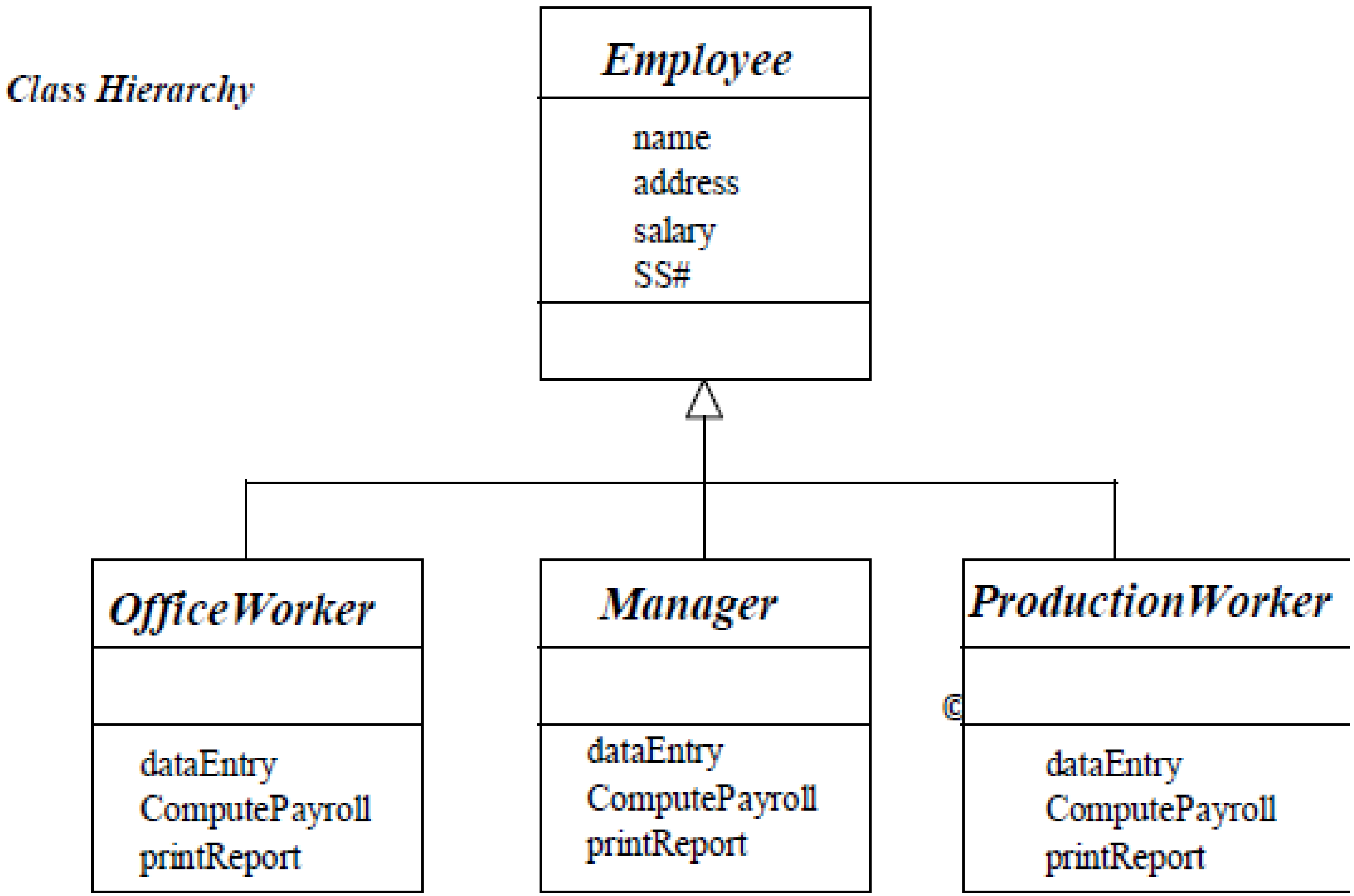
- Models "kind of" hierarchy
- Powerful notation for sharing similarities among classes while preserving their differences
- UML Notation: An **arrow with a triangle**



Generalization

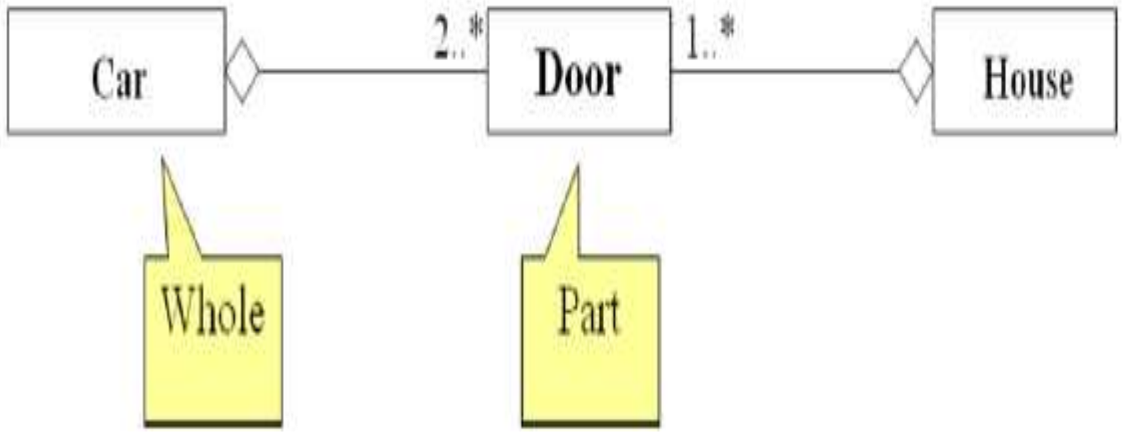


Generalization



Aggregation

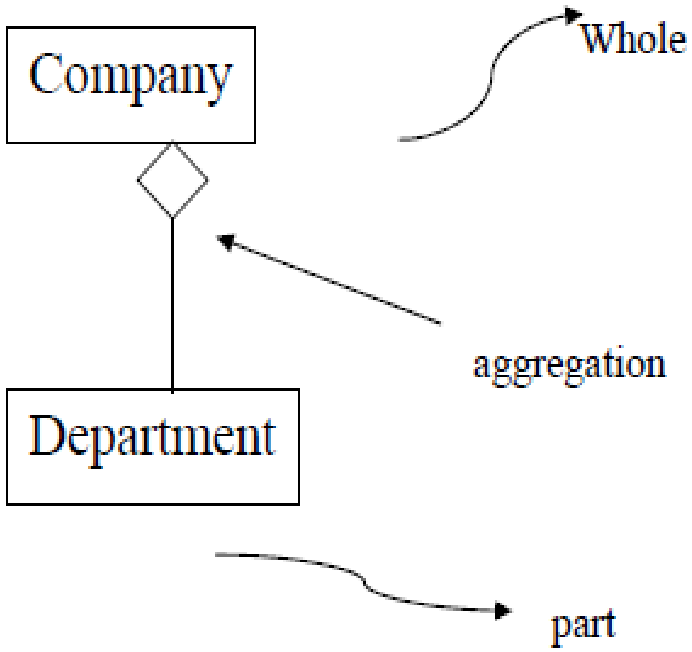
A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.



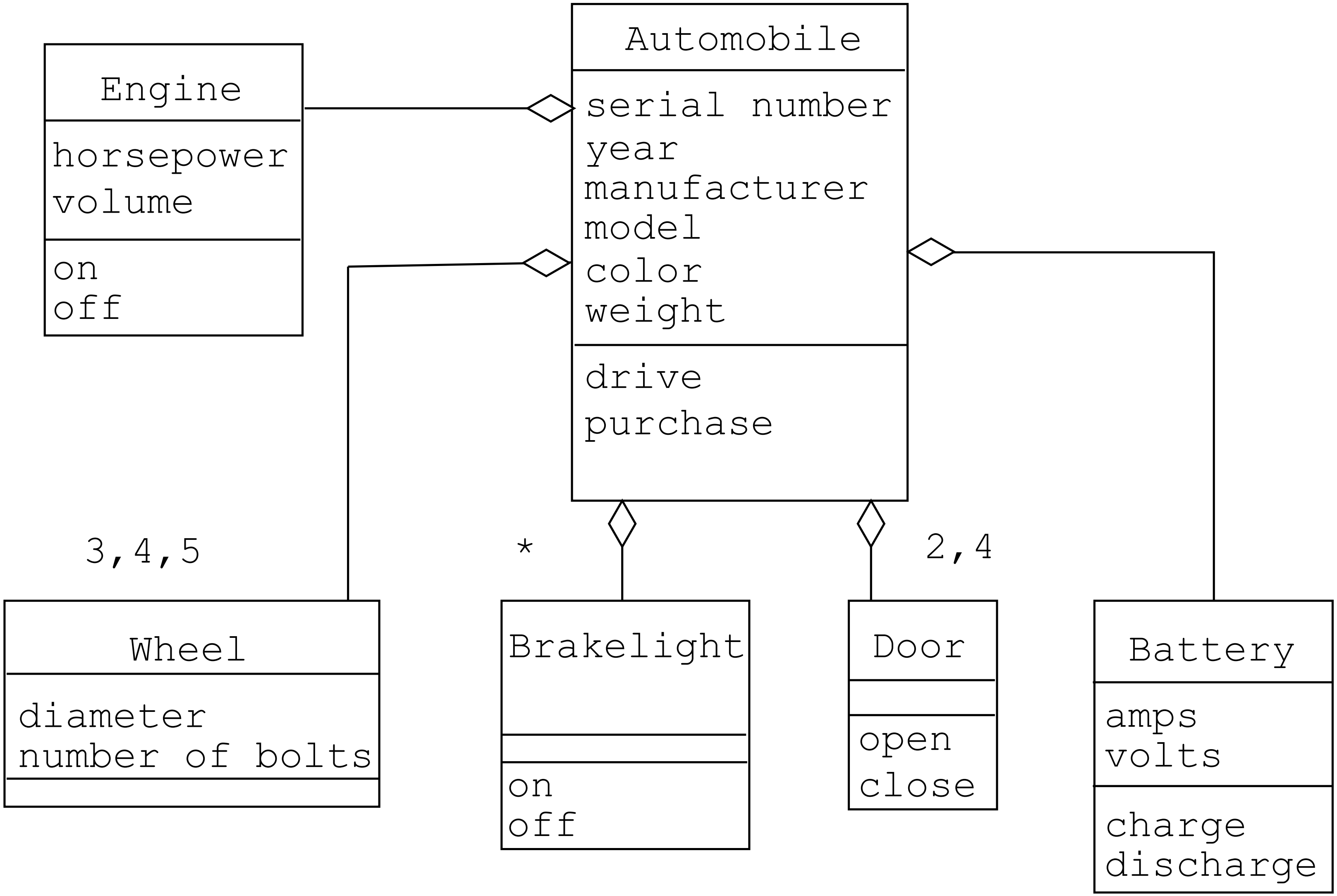
Aggregation is also called **whole-part or HasA.** For example, an **Aircraft contains an Engine** or in other words**, an Aircraft has an Engine.**

Aggregation

- Represents a “has-a” (whole-part) relationship
- An object of the whole has objects of the part



Aggregation



Generalization Vs. Aggregation

- Both associations describe trees (hierarchies)
  - Aggregation tree describes a-part-of relationships (also called **and-relationship, Has –a Relationship, containership**)
  - Inheritance tree describes "kind-of" relationships (also called or-relationship, is-a)
- Aggregation **relates instances** (involves two or more *different* **objects**)
- **Inheritance relates classes** (a way to structure the description of a *single* object)

Object Composition

- **Use of objects in a class as data members** is referred to as object composition
- Object can be a collection of many other objects
- The relationship is called a **has-a relationship** or **containership**
- When it comes to the real world programming problem, an object of class TextBox can be contained in the class Form and can so be said as a Form contains a TextBox (or in another way, a Form is composed of a TextBox).
- Inheritance represents **'is -a ' relationship** in OOP, while Containership represents a' **has -a' relationship.**

Containership

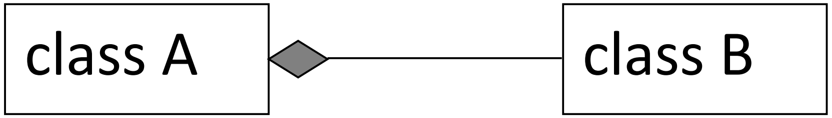
•*Containment or containership:* This relationship is applied when the part contained with in the whole part, dies when the whole part dies.

It is represented as darked diamond at the whole part.

example:

```
class A{
//some code
};
class B
{
A aa; // an object of class A;
// some code for class B;
};
```

In the above example we see that an object of class A is instantiated with in the class B. so the object class A dies when the object class B dies. We can represent it in diagram like this.



Containership

```
class address {
        int hno;
        char colony[20];
        char dist[20];
        char state[20];
        int pincode;
        public:
        void get_data();
        void show_data();
       };
class person {
       char name[20];
       address resadd;
       };
```

Containership

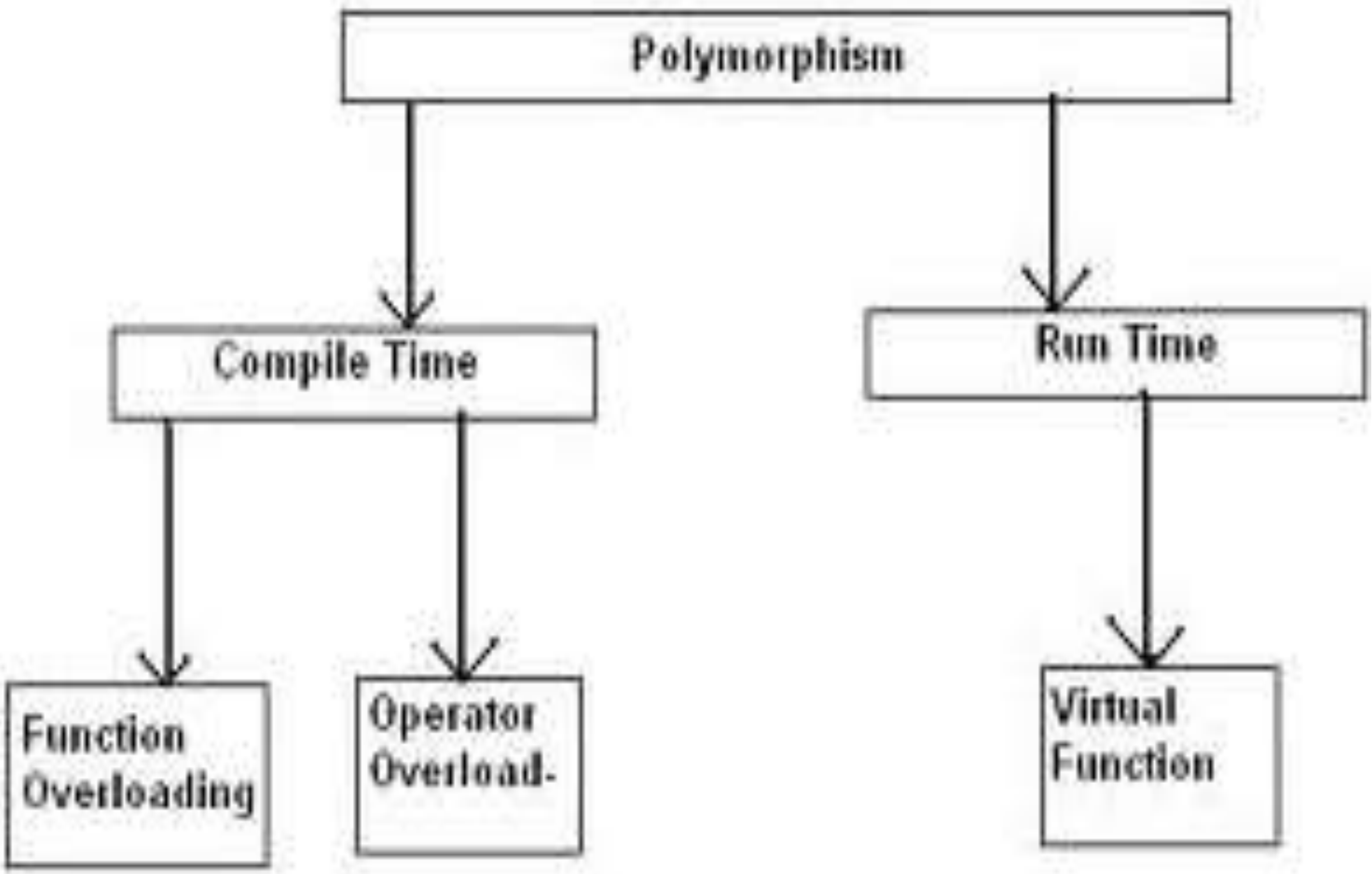
- Here, Containership (resadd is a new variable / object of class address). The above declaration establishes the relationship i.e. **A person "has an" address. Now an object of a class person will always contain an object of class address.**
- To invoke the function of contained object, **resadd.getdata(), resadd.showdata() will be mentioned in the program.**
- Inheritance and Containership two important concepts found in OOP (Object Orientated Programming: Example- C++).

Containership

```
class Department
{
protected:
int id;
char name[50];
public:
void setDepartment()
{
cout<>id;
cout<>name;
}
void displayDepartment()
{
cout<<"\n Department ID is:"<<id<<"\n";
cout<<"\n Department Name
is:"<<name<<"\n";
}};
```

```
class Employee
{
protected:
int eid;
char ename[50];
Department dobj;
public:
void setEmployee()
{
cout<>eid;
cout<>ename;
dobj.setDepartment();
}
void displayEmployee()
{
cout<<"\n Employee ID is:"<<eid<<"\n";
cout<<"\n Employee Name is:"<<ename<<"\n";
dobj.displayDepartment();
}
};
```

POLYMORPHISM



POLYMORPHISM / OVERLOADING

- A Greek term suggest the ability to take more than one form.
- It is a property by which the same message can be sent to the objects of different class.

  Example: Draw a shape (Box, Triangle, Circle etc.), Move (Chess, Traffic, Army).
- Allows to create multiple definition for operators & functions.

  Example: ‘+’ is used for adding numbers / to concatenate two string / Sets of Union and so on.
- There are two types of polymorphism, compile time polymorphism and run time polymorphism. It is also known as early or static binding and run time binding.

POLYMORPHISM (Contd.)

- Function and operator overloading is the example of compile time polymorphism and virtual function is the example of run time polymorphism.
- A Virtual function, equated to zero is called pure virtual function.
- Dynamic Binding/ Late Binding. Run-time dependent. Execution depends on the base of a particular definition.
- Extensively used in implementing inheritance.

POLYMORPHISM / OVERLOADING

- A Greek term suggest the **ability to take more than one form.**
- Typically occurs when there is a hierarchy of classes related by inheritance.
- **Simply implies that call to a member function will cause a different object to be executed, depending on the type of object that invokes the function.**
- **Instead of inventing a new name for each new function you add to a program the same names can be reused.**
- It is a property by which the same message can be sent to the objects of different class.

  Example: Draw a shape (Box, Triangle, Circle etc.), Move (Chess, Traffic, Army).

Compile time polymorphism

- Involves binding of functions based on the
  - **number of arguments,**
  - **type of arguments**
  - **sequence of arguments.**
- **This information is known to the compiler at compile time. So compiler selects the appropriate function for a particular call at compile time itself.**
- **The various parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time.**
- **This form of association is called <u>early binding</u>. The term early binding implies that when the program is executed, the calls are already bound to the appropriate functions.**

Runtime Polymorphism

- Refers to an entity changing its form depending on circumstances.
- A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed.
- **The term *late binding* refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.**

Function Overloading

Function overloading is a concept where several function declarations are specified with a single and a same function name within the same scope. Such functions are said to be overloaded. C++ allows functions to have the same name. Such functions can only be distinguished by their number and type
of arguments.

Example:

```
float divide(int a , int b);
float divide(float a, float b);
```

The function **divide()**, which takes two integer inputs, is different from the function **divide()** which takes two float inputs.

What is the need for function overloading?

Every object has characteristics and associated behavior. An object may behave differently with change in its characteristics. Therefore, in order to simulate real world objects in programming environment, it is necessary to have function overloading.

For Example:

```
float AddNumber(float a, float b)
{
            return a + b ;
}
int AddNumber(int a, int b)
{
            return a + b ;
}
void main()
{
            cout<<AddNumber( 21, 36 ) ;
            cout<<AddNumber( 6.72, 2.22 ) ;
}
```

Function overloading not only implements polymorphism but also reduces number of comparisons in a program and thereby making the program run faster.

How to implement function overloading?

The key to function overloading is the function's argument list which is also known as function signature. It is the signature and not the function type that enables function overloading.

If two functions have the same number and type of arguments in the same order, they are said to have the same signature.

```
void abc(int a, float b)<br>
void abc(int x, float y)
```

Both these functions have the same signature.

Contd…

Sample for Function Overloading

C++ allows you to overload a function provided the function has the same name but different signatures. The signature can differ in the number of arguments or in the type of arguments, or both. To overload a function, all you need to do is, declare and define all the functions with the same name but different signatures.

Example:

```
void prnsqr(int i);
void prnsqr(char c);
void prnsqr(float f);
void prnsqr(double d);

void prnsqr(int i)
{
        cout<<"\n Integer "<< i<<"'s square is "<<i*i<<"\n";
}
void prnsqr(char c)
{
        cout<<"\n Character "<<c <<" thus no square "<<"\n";
}
void prnsqr(float f)
{
        cout<<"\n Float "<<f <<"'s square is "<<f*f<<"\n";
}
void prnsqr(double d)
{
        cout<<"\n Double "<<d<<"'s square is "<<d*d<<"\n";
}
```

Contd…

When a function, with same name, is declared more than once in the program, the compiler will interpret the second declaration as follows:

- If the signature of subsequent function matches the previous function, then the second is treated as the re-declaration of the first.
- If the signature of both the functions match exactly, but the return type differs, then the second declaration is treated as an erroneous re-declaration of the first and is flagged at compile time as an error.

For example,

```
float square(float f);
double square(float x); //error
```

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Class member access operators (., .*).
- Scope resolution operator (::).
- Size operator (**sizeof**).
- Conditional operator (?:).

BPIT

C++ Operators Overloading

	Operator	Type
Binary Operator	+, -, *, /, %	Arithmetic Operators
	<,<=, >, >=, ==, !=	Relational Operators
	&&, ||, !	Logical Operators
	&, |, <<, >>, ~, ^	Bitwise Operators
	=, +=, -=,*=, /=, %=	Assignment Operators
Unary Operator	++, --	Unary Operator
Ternary Operator	?:	Ternary or Conditional Operator

C++ Operators Overloading

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.
class_name is the name of the class.
operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

C++ Operators Overloading

```
class class_name
{
    ... .. ...
    public
       return_type operator symbol (argument(s))
       {
           ... .. ...
       }
    ... .. ...
};
```

Here is an explanation for the above syntax:

- The return_type is the return type for the function.
- Next, you mention the operator keyword.
- The symbol denotes the operator symbol to be overloaded. For example, +, -, <, ++.
- The argument(s) can be passed to the operator function in the same way as functions.

C++ Operators Overloading

Overloaded operator functions can be invoked by expressions such as

```
op x or x op
```

for unary operators and

```
x op y
```

for binary operators. *op* x (or x *op*) would be interpreted as

```
operator op (x)
```

for **friend** functions. Similarly, the expression x op y would be interpreted as either

```
x.operator op (y)
```

in case of member functions, or

```
operator op (x,y)
```

in case of **friend** functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

```
#include <iostream>

using namespace std;

class space
{
        int x;
        int y;
        int z;
 public:
        void getdata(int a, int b, int c);
        void display(void);
        void operator-();      // overload unary minus
};
void space :: getdata(int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void space :: display(void)
{
        cout << x << " ";
        cout << y << " " ;
        cout << z << "\n";
}
void space :: operator-()
{
        x = -x;
        y = -y;
        z = -z;
}

int main()
{
        space S;
        S.getdata(10, -20, 30);
```

C++ Unary Operator Overloading

```
    cout << "S : ";
    S.display();

    -S;                    // activates operator-() function

    cout << "S : ";
    S.display();

    return 0;
}
```

PROGRAM 7.1

The Program 7.1 produces the following output:

```
S : 10 -20 30
S : -10 20 -30
```

C++ Unary Operator (++) Overloading

```
#include <iostream>
using namespace std;
class Test
{
  private:
     int num;
  public:
     Test(): num(8){}
     void operator ++()       {
        num = num+2;
     }
     void Print() {
         cout<<"The Count is: "<<num;
     }
};
```

```
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Output

The Count is: 10

Program to overload the binary operators.

```
#include <iostream>
using namespace std;
class A
{

    int x;
      public:
      A(){}
    A(int i)
    {
       x=i;
    }
    void operator+(A);
    void display();
};
```

```
void A :: operator+(A a)
{

    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Output

The result of the addition of two objects is : 9

7.4 Overloading Binary Operators

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. In Chapter 6, we illustrated, how to add two complex numbers using a friend function. A statement like

```
C = sum(A, B);            // functional notation.
```

was used. The functional notation can be replaced by a natural looking expression

```
C = A + B;                // arithmetic notation
```

by overloading the + operator using an operator+() function. The Program7.2 illustrates how this is accomplished.

OVERLOADING + OPERATOR

```
#include <iostream>

using namespace std;

class complex
{
        float x;                              // real part
        float y;                              // imaginary part
  public:
        complex(){ }                          // constructor 1
        complex(float real, float imag)       // constructor 2
        { x = real; y = imag; }
        complex operator+(complex);
        void display(void);
};
```

Program to overload the binary operators.

```
complex complex :: operator+(complex c)
{
        complex temp;               // temporary
        temp.x = x + c.x;           // these are
        temp.y = y + c.y;           // float additions
        return(temp);
}


void complex :: display(void)
{
        cout << x << " + j" << y << "\n";
```

Program to overload the binary operators.

```
}

int main()
{
        complex C1, C2, C3;          // invokes constructor 1
        C1 = complex(2.5, 3.5);      // invokes constructor 2
        C2 = complex(1.6, 2.7);
        C3 = C1 + C2;

        cout << "C1 = "; C1.display();
        cout << "C2 = "; C2.display();
        cout << "C3 = "; C3.display();

        return 0;
}
```

PROGRAM 7.2

The output of Program 7.2 would be:

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

Virtual function

C++ virtual function is,

- **A member function of a class**
- Declared with ***virtual*** **keyword**
- Usually has **a different functionality in the derived class**
- A **function call is resolved at run-time**

- The difference between a non-virtual c++ member function and a virtual member function is, the **non-virtual member functions are resolved at compile time.** This mechanism is called ***static binding***.

- Where as the c++ **virtual member functions are resolved during run-time. This mechanism is known as *dynamic binding*.**

Pointer object of base class can point to any object of derived class but reverse is not true.

Virtual Function

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform Late Binding on this function.

Virtual Keyword is used to make a member function of the base class Virtual.

Late Binding in C++

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime** Binding.

Virtual Function

Problem without Virtual Keyword

Let's try to understand what is the issue that virtual keyword fixes,

```
class Base
{
   public:
   void show()
   {
      cout << "Base class";
   }
};

class Derived:public Base
{
   public:
   void show()
   {
      cout << "Derived Class";
   }
}
```

```
int main()
{
   Base* b;       //Base class pointer
   Derived d;     //Derived class object
   b = &d;
   b->show();     //Early Binding Occurs
}
```

Output-
Base class

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

Virtual Function

Using Virtual Keyword in C++

We can make base class's methods virtual by using virtual keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

```
class Base
{
    public:
    virtual void show()
    {
        cout << "Base class\n";
    }
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Derived Class";
    }
}
```

```
int main()
{
    Base* b;       //Base class pointer
    Derived d;     //Derived class object
    b = &d;
    b->show();     //Late Binding Occurs
}
```

Output-
Derived class

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

Virtual Function

Using Virtual Keyword and Accessing Private Method of Derived class

We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

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

class A
{
   public:
   virtual void show()
   {
      cout << "Base class\n";
   }
};

class B: public A
{
   private:
   virtual void show()
   {
      cout << "Derived class\n";
   }
};
```

```
int main()
{
   A *a;
   B b;
   a = &b;
   a->show();
}
```

Output-
Derived class

Virtual Function

```
// CPP program to illustrate  concept of Virtual Functions
 #include <iostream>
using namespace std;

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
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}
```

Output-

print derived class

show base class

Virtual Function

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is print derived class as pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time(output is show base class as pointer is of base type).

When we use the same function name in both the base and derived classes, the function in base class is declared as *virtual* using the keyword **virtual** preceding its normal declaration. When a function is made **virtual**, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function. Program 9.12 illustrates this point.

VIRTUAL FUNCTIONS

```
#include <iostream>

using namespace std;

class Base
{
  public:
        void display() {cout << "\n Display base ";}
        virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
  public:
        void display() {cout << "\n Display derived";}
        void show() {cout << "\n show derived";}
};

int main()
{
        Base B;
        Derived D;
        Base *bptr;

        cout << "\n bptr points to Base \n";
        bptr = &B;
        bptr -> display();    // calls Base version
        bptr -> show();       // calls Base version

        cout << "\n\n bptr points to Derived\n";
        bptr = &D;
        bptr -> display();    // calls Base version
        bptr -> show();       // calls Derived version

        return 0;
}
```

PROGRAM 9.12

Virtual Function

The output of Program 9.12 would be:

```
bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived
```

note

When **bptr** is made to point to the object **D**, the statement

```
bptr -> display();
```

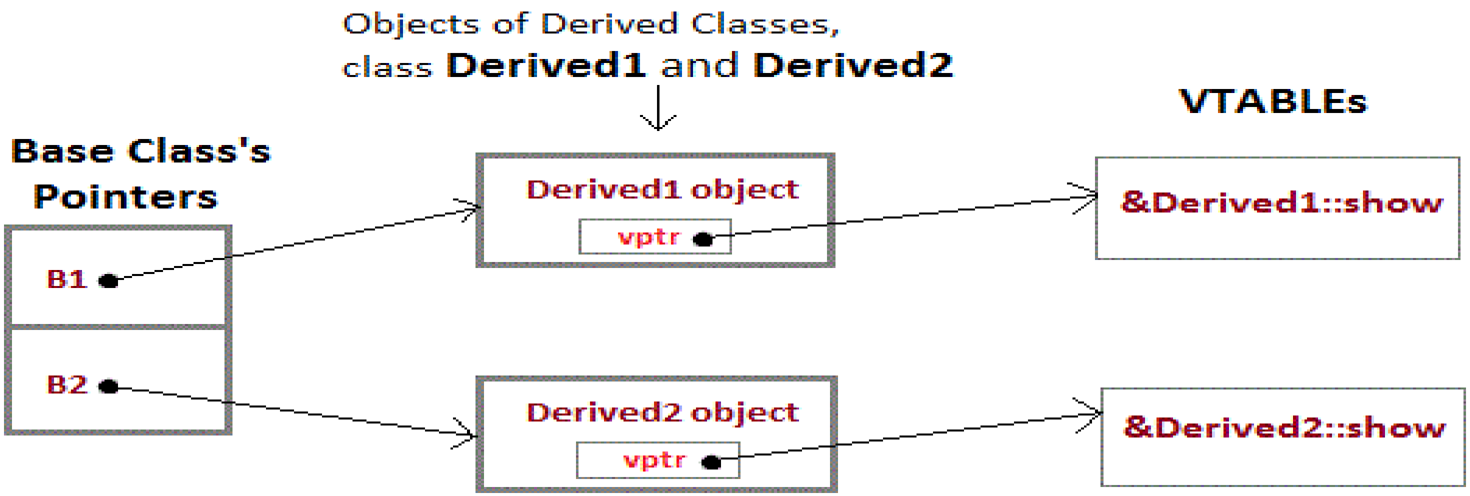
calls only the function associated with the **Base** (i.e. **Base :: display**( )), whereas the statement

```
bptr -> show();
```

calls the **Derived** version of **show()**. This is because the function **display()** has not been made **virtual** in the **Base** class.

One important point to remember is that, we must access **virtual** functions through the use of a pointer declared as a pointer to the base class. Why can't we use the object name (with the dot operator) the same way as any other member function to call the virtual functions?. We can, but remember, run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Mechanism of Late Binding in C++



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

To accomplish late binding, Compiler creates VTABLEs, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called vpointer, pointing to VTABLE for that object. Hence when function is called, compiler is able to resovle the call by binding the correct function using the vpointer.

Virtual Function

Important Points to Remember

- Only the Base class Method's declaration needs the Virtual Keyword, not the definition.
- If a function is declared as virtual in the base class, it will be virtual in all its derived classes.
- The address of the virtual Function is placed in the VTABLE and the copiler uses VPTR(vpointer) to point to the Virtual Function.

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

virtual void display() = 0;

Pure Virtual Function

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show() = 0;
};
class Derived : public Base
{
   public:
   void show()
   {
      std::cout << "Derived class is derived from the base class." << std::endl;
   }
};
```

```
int main()
{
   Base *bptr;
   //Base b;
   Derived d;
   bptr = &d;
   bptr->show();
   return 0;
}
```

Output:
Derived class is derived from the base class.

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

Pure virtual (abstract) functions and abstract base classes

C++ allows you to create a special kind of virtual function called a **pure virtual function** (or **abstract function**) that has no body at all! A pure virtual function simply acts as **a placeholder that is meant to be redefined by derived classes.**

A **pure virtual function** is a function that has ***the notation "= 0"*** in the declaration of that function.

class SomeClass

{

public:

virtual void pure_virtual() = 0; // a pure virtual function // note that there is no function body

};

pure virtual function

A pure virtual function can have an implementation in C++ – which is something that even many expert C++ developers do not know.

class SomeClass

{

public:

virtual void pure_virtual() = 0; // a pure virtual function // note that there is no function body };

/*This is an implementation of the pure_virtual function which is declared as a pure virtual function. This is perfectly legal: */

void SomeClass::pure_virtual()

{

cout<<"This is a test"<<endl;

}

pure virtual function

•It **is rare to see a pure virtual function with an implementation in real-world code**, but having that implementation may be desirable when you think that classes which derive from the base class may need some sort of **default** behavior for the pure virtual function.

•So, for example, **if we have a class that derives from our SomeClass class above, we can write some code like this** – where the derived class actually makes a call to the pure virtual function implementation that is inherited:

pure virtual function

```
class base
{public:
virtual void show()=0; //pure
virtual function
};
class derived1 : public base
{public:
void show(){
cout<<"\n Derived 1";}};
class derived2 : public base
{
public:
void show()
{cout<<"\n Derived 2";}};
```

```
void main()
{
base *b; derived1 d1; derived2 d2;
b = &d1;
b->show();
b = &d2;
b->show();
}
```

Inner Working of Virtual Functions

- Don’t need to know how to use it!
  - Principle of information hiding
- Virtual function table
  - Compiler creates it
  - Has pointers for each virtual member function
  - Points to location of correct code for that function
- Objects of such classes also have pointer
  - Points to virtual function table

Drawback of Virtual Function

- Calling a virtual function is **slower** than calling a non-virtual function for a couple of reasons:
- First, we have **to use the *__vptr to get to the appropriate virtual table.**
- Second, we have to **index the virtual table to find the correct function to call. Only then can we call the function.**
- As a result, **we have to do 3 operations to find the function to call,** as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call.
- However, with modern computers, this added time is usually fairly insignificant/unimportant.

The virtual table

•To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The **virtual table** is a **lookup table of functions used to resolve function calls** in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as **"vtable", "virtual function table", "virtual method table", or "dispatch table".**

•First, **every class** that uses virtual functions (or is derived from a class that uses virtual functions) is **given it's own virtual table**. This table is simply a **static array** that the **compiler sets up at compile time. A virtual table contains one entry for each virtual function** that can be called by objects of the class.

The virtual table

•Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

•Second, the **compiler also adds a hidden pointer to the base class,** which we will call *__vptr. *__vptr is set (automatically) ***when a class instance is created so that it points to the virtual table for that class.***

The virtual table

```
class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};
 class D1: public Base
{
public:
    virtual void function1() {};
};
 class D2: public Base
{
public:
    virtual void function2() {};
};
```

Because there are **3 classes here**, the compiler will set up **3 virtual tables: one for Base, one for D1, and one for D2.**

The **compiler also adds a hidden pointer to the most base class** that uses virtual functions. Although the **compiler does this automatically,**

The virtual table

we'll put it in the next example just to show where it's added:

```
class Base
{
public:
   FunctionPointer *__vptr;
   virtual void function1() {};
   virtual void function2() {};
};
 class D1: public Base
{
public:
   virtual void function1() {};
};
 class D2: public Base
{
public:
   virtual void function2() {};
};
```

The virtual table

- **When a class object is created, *__vptr is set to point to the virtual table for that class.** For example, **when a object of type Base is created, *__vptr is set to point to the virtual table for Base.** When objects of type D1 or D2 are constructed, *__vptr is set to point to the virtual table for D1 or D2 respectively.

- Now, let's talk about how these virtual tables are filled out. Because **there are only two virtual functions here, each virtual table will have two entries (one for function1(), and one for function2()).** Remember that when these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.

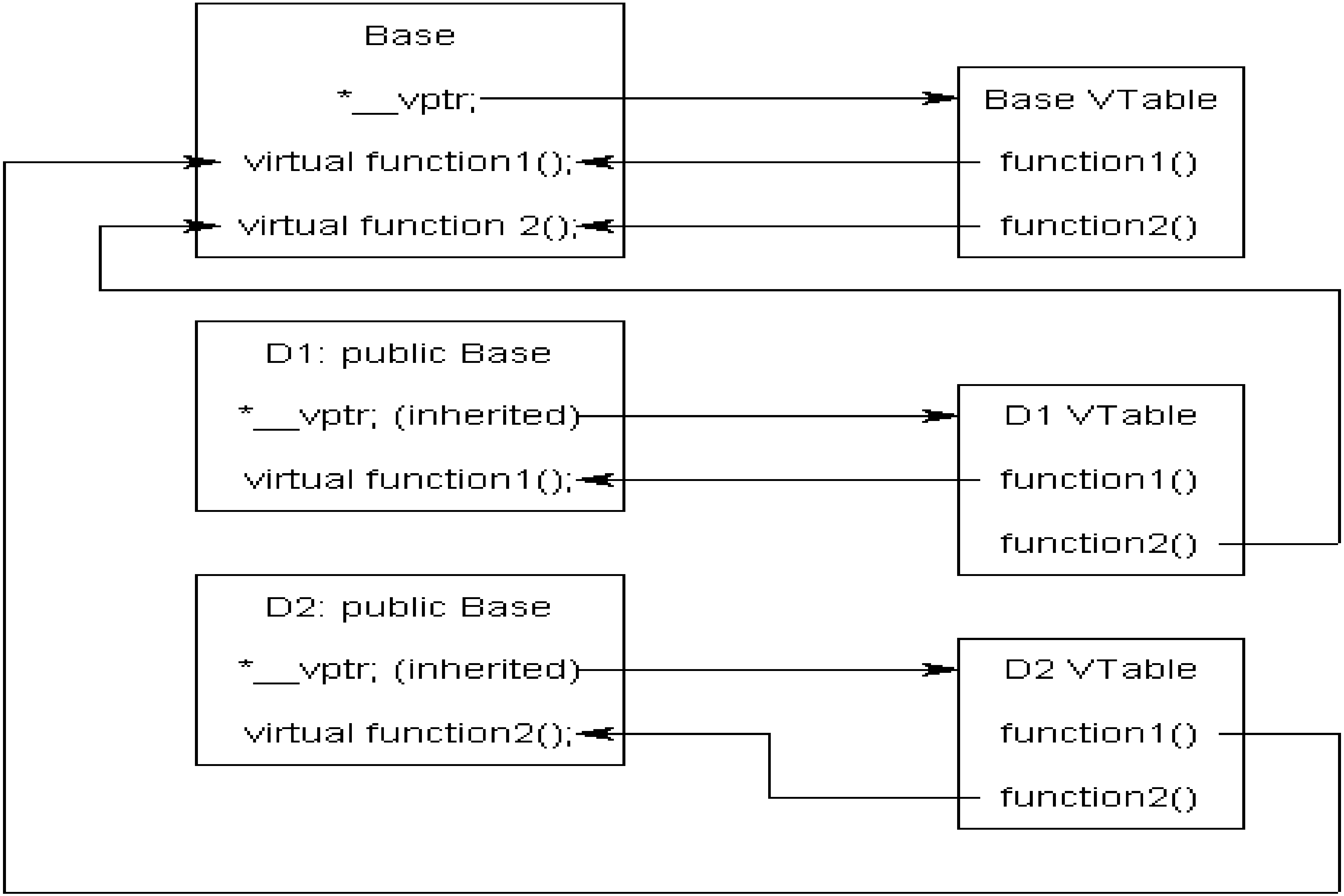
The virtual table

•Base’s virtual table is simple. An object of type Base can only access the members of Base. Base has no access to D1 or D2 functions. **Consequently, the entry for function1 points to Base::function1(), and the entry for function2 points to Base::function2().**

•**D1′s virtual table** is slightly more complex. An object of type D1 can access members of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn’t overridden function2(), so the entry for function2 will point to Base::function2().

•D2′s virtual table is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().

Here's a picture of this graphically:



Overriding in C++

- Overriding: a method in a parent class is replaced in a child class by a method having exact same type signature.
- In C++, overriding uses the keyword ***virtual***.
- When a class contains a virtual method, an internal table called the ***virtual method table*** is created.
- This table is used in the implementation of the dynamic binding of message to method required by the virtual method.
- Each instance of a class must contain an additional hidden pointer value, which references the virtual method table.
- A pure virtual method must be overridden in subclasses.

Abstract Classes

- Abstract classes act as expressions of general concepts from which more specific classes can be derived. You cannot create an object of an abstract class type; however, you can use pointers and references to abstract class types.
- A class that contains at least one pure virtual function is considered an abstract class. Classes derived from the abstract class must implement the pure virtual function or they, too, are abstract classes.
- A virtual function is declared as "pure" by using the *pure-specifier* syntax

```
class Account { public: Account( double d ); // Constructor.
virtual double GetBalance(); // Obtain balance.
virtual void PrintBalance() = 0; // Pure virtual function.
private: double _balance; };
```

Abstract Classes

```
class Animal {
    public:  virtual void speak() = 0;};
class Bird : public Animal {
    public:  virtual void speak() { printf("twitter"); }};
class Mammal : public Animal {
    public:  virtual void speak() { printf("can't speak"); }
             void bark() { printf("can't bark"); }};
class Cat : public Mammal {
    public:  void speak() { printf("meow"); }
             virtual void purr() { printf("purrrrr"); }};
class Dog : public Mammal {
    public:  virtual void speak() { printf("wouf"); }
             void bark() { printf("wouf"); }
};
```

Restrictions on using Abstract Classes

- Abstract classes cannot be used for:
- Variables or member data
- Argument types
- Function return types
- Types of explicit conversions
- Another restriction is that if the constructor for an abstract class calls a pure virtual function, either directly or indirectly, the result is undefined. However, constructors and destructors for abstract classes can call other member functions.
- Pure virtual functions can be defined for abstract classes, but they can be called directly only by using the syntax:
- ***abstract-class-name* :: *function-name*( )**
- This helps when designing class hierarchies whose base class(es) include pure virtual destructors, because base class destructors are always called in the process of destroying an object.

Order of invocation of constructor

Method of Inheritance	Order of Execution
Class D:public B{…..};	· B(): base constructor; · D(): derived constructor
class D:public B1, public B2{…..};	• B1(): base constructor; • B2(): base constructor; • D(): derived constructo
class D:public B1, virtual B2{…..};	• B2(): virtual base cons.; • B1(): base constructor; • D(): derived constructor
class D1:public B{….}; class D2:public D1{…};	• B(): super base constructor; • D1(): base constructor; • D2() derived constructor

Destructors in Derived Classes

- Invoked in reverse order of the constructor invocation

```
#include <iostream.h>
class B1{
    public :
        B1(){cout<< "no argument constructor in B1";}
        ~B1(){cout<< "destructor in B1";}
};
class B2{
    public :
        B2(){cout<< "no argument constructor in B2";}
        ~B2(){cout<< "destructor in B2";}
};
```

Destructors in Derived Classes

```
class D: public B1, public B2
{
        public :
                D(){cout<< "no argument constructor in D";}
                ~D(){cout<< "destructor in D";}
};
void main(){ D objd;}
```

Run

no argument constructor in B1
no argument constructor in B2
no argument constructor in D
destructor in D
destructor in B2
destructor in B1

in C++ a **destructor is generally used to deallocate memory** and do some other cleanup for a class object and it's class members whenever an object is destroyed.

Example *without* a Virtual Destructor:

```
#include iostream.h
class Base
{
public:
Base()
{
 cout<<"Constructing Base";
}
 // this is a destructor:
 ~Base()
{
cout<<"Destroying Base";
}
};
```

```
class Derive: public Base
{
public:
Derive()
{
 cout<<"Constructing Derive";
}
~Derive(){
cout<<"Destroying Derive";
}
};
void main()
{
Base *basePtr = new Derive();
delete basePtr;
}
```

o/p–
Constructing Base
Constructing Derive
Destroying Base

we can see that the constructors get called in the appropriate order when we create the **Derive class object pointer** in the main function.
But there is a major problem with the code above: the **destructor for the "Derive" class does not get called** at all when we delete ‘basePtr’.

what we can do is make the base class destructor virtual, and that will ensure that the destructor for any class that derives from Base (in our case, its the "Derive" class) will be called.

```
class Base
{
 public:
Base()
{
 cout<<"Constructing Base";
}
// this is a destructor:
virtual ~Base()
{
cout<<"Destroying Base";
}
};
```

o/p–

Constructing Base

Constructing Derive

Destroying Derive

Destroying Base

Conclusion

- Inheritance provides the concept of reusability. The derived class inherits some or all of the properties of the base class.
- A private member of a class cannot be inherited either in public mode or in private mode.
- The member functions of a derived class can directly access only the in the protected and public data.
- Multipath inheritance may lead to duplication of inherited members from a "grandparent" base class. This may be avoided making the common base class a virtual base class.
- In multiple inheritance, the base classes are constructed in the order in which they in the declaration of the derived class.
- In multilevel inheritance, the constructors are executed in the order of inheritance.
- A class can contain object of other classes. This is known as containership or nesting.

Generic Programming

Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character.

The advantages of Generic Programming are

- Code Reusability
- Avoid Function Overloading
- Once written it can be used for multiple times and cases.

Generics can be implemented in C++ using Templates. Template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

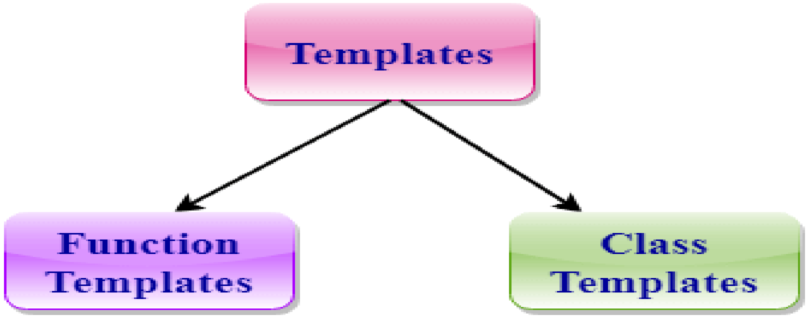
Generic Functions using Template:

We write a generic function that can be used for different data types.

Examples of function templates are sort(), max(), min()

Templates can be represented in two ways:

- Function templates
- Class templates



Function Templates:

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Function Template

Syntax of Function Template

```
template < class Ttype>
ret_type func_name(parameter_list)
{
   // body of function.
}
```

Where Ttype: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

12.4 Function Templates

Like class templates, we can also define function templates that could be used to create a family of functions with different argument types. The general format of a function template is:

```
template<class T>
returntype functioname (arguments of type T)
{
        // .....
        // Body of function
        // with type T
        // wherever appropriate
        // .....
}
```

The function template syntax is similar to that of the class template except that we are defining functions instead of classes. We must use the template parameter **T** as and when necessary in the function body and in its argument list.

The following example declares a **swap()** function template that will swap two values of a given type of data.

```
template<class T>
void swap(T&x, T&y)
{
   T temp = x;
   x = y;
   y = temp;
}
```

This essentially declares a set of overloaded functions, one for each type of data. We can invoke the **swap()** function like any ordinary function. For example, we can apply the **swap()** function as follows:

```
void f(int m,int n,float a,float b)
{
     swap(m,n);          // swap two integer values
     swap(a,b);          // swap two float values
     // .....
```

This will generate a **swap()** function from the function template for each set of argument types. Program 12.4 shows how a template function is defined and implemented.

FUNCTION TEMPLATE - AN EXAMPLE

```
#include <iostream>

using namespace std;

template <class T>
void swap(T &x, T &y)
{
        T temp = x;
        x = y;
        y = temp;
};

void fun(int m,int n,float a,float b)
{
        cout << "m and n before swap: " << m << " " << n << "\n";
        swap(m,n);
        cout << "m and n after swap:  " << m << " " << n << "\n";

        cout << "a and b before swap: " << a << " " << b << "\n";
        swap(a,b);
        cout << "a and b after swap:  " << a << " " << b << "\n";
}

int main()
{
        fun(100,200,11.22,33.44);

        return 0;
}
```

PROGRAM 12.4

The output of Program 12.4 would be:

```
m and n before swap:    100 200
m and n after swap:     200 100
a and b before swap:    11.22 33.439999
a and b after swap:     33.439999 11.22
```

Another function often used is **sort()** for sorting arrays of various types such as **int** and **double**. The following example shows a function template for bubble sort:

Function Template

Let's see a simple example of a function template:

```
#include <iostream>
using namespace std;
template<class T>
T add(T &a,T &b)
{
    T result = a+b;
    return result;

}
```

```
int main()
{
  int i =2;
  int j =3;
  float m = 2.3;
  float n = 1.2;
  cout<<"Addition of i and j is :"<<add(i,j);
  cout<<'\n';
  cout<<"Addition of m and n is
:"<<add(m,n);
  return 0;
}
```

Output:

Addition of i and j is :5

Addition of m and n is :3.5

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
 template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
   // body of function.
}
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Function Templates with Multiple Parameters

```
#include <iostream>
using namespace std;
template<class X,class Y>
void fun(X a,Y b)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}

int main()
{
   fun(15,12.3);

   return 0;
}
```

Output:

Value of a is : 15
Value of b is : 12.3

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

Let's understand this through a simple example:

```
#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
    std::cout << "Value of a is : " <<a<< std::endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
    std::cout << "Value of b is : " <<b<< std::endl;
    std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
   fun(10);
   fun(20,30.5);
   return 0;
}
```

Output:

Value of a is : 10
Value of b is : 20
Value of c is : 30.5

How VPTR and Virtual table works(Memory Layout)

```
class Test
{
public:
int data1;
int data2;
int fun1();
};
int main()
{
Test obj;
cout << "obj's Size = " << sizeof(obj) << endl;
cout << "obj 's Address = " << &obj << endl;

return 0;
```

OUTPUT:

Sobj's Size = 4 Bytes

obj 's Address = 0012FF7C

Note: Any **Plane member function does not take any memory.**

Example 2: Memory Layout of Derived class

```
class Test
{
public:
int a;
int b;
};
class dTest : public Test
{
public:
int c;
};
int main()
{
Test obj1;
cout << "obj1's Size = " << sizeof(obj1) << endl;
cout << "obj1's Address = " << &obj1 << endl;
dTest obj2;
cout << "obj2's Size = "<< sizeof(obj2) << endl;
cout << "obj2's Address = "<< &obj2 << endl;
return 0;
}
```

OUTPUT:

obj1's Size = 4

obj1's Address = 0012FF78

obj2's Size = 6

obj2's Address = 0012FF6C

Example 3: Memory layout If we have one virtual function.

```
class Test
{
public:
int data;
virtual void fun1()
{
cout << "Test::fun1" << endl;
}
};
int main()
{
Test obj;
cout << "obj's Size = " << sizeof(obj) << endl;
cout << "obj's Address = " << &obj << endl;
return 0;
}
```

OUTPUT:

obj's Size = 4

obj's Address = 0012FF7C

Note: Adding **one virtual function in a class takes 2 Byte extra.**

Example 4: More than one Virtual function

```
class Test
{
public:
int data;
virtual void fun1() { cout << "Test::fun1" << endl; }
virtual void fun2() { cout << "Test::fun2" << endl; }
virtual void fun3() { cout << "Test::fun3" << endl; }
virtual void fun4() { cout << "Test::fun4" << endl; }
};
int main()
{
Test obj;
cout << "obj's Size = " << sizeof(obj) << endl;
cout << "obj's Address = " << &obj << endl;
return 0;
}
```

Example 4: More than one Virtual function

OUTPUT:

obj's Size = 4

obj's Address = 0012FF7C

Note: **Adding more virtual functions in a class, no extra size taking** i.e. **Only one machine size taking(i.e. 2 byte)**

Multiple Inheritance

```
class Base1
{
public:
virtual void fun();
};
class Base2
{
public:
virtual void fun();
};
class Base3
{
public:
virtual void fun();
};
class Derive : public Base1, public Base2, public Base3
{
};
int main()
{
Derive obj;
cout << "Derive's Size = " << sizeof(obj) << endl;
return 0;
}
```

OUTPUT:
Derive's Size = 6

Operator Returns Values

Conclusion

- Polymorphism simple means one name having multiple forms.
- There are two types of polymorphism, compile time polymorphism and run time polymorphism. It is also known as early or static binding and run time binding.
- Function and operator overloading is the example of compile time polymorphism and virtual function is the example of run time polymorphism.
- A Virtual function, equated to zero is called pure virtual function.

Review Questions [Objective Types]

1. Assume a class D is privately derived from class B type of members can an object of class D locate in main() access?
2. When does an ambiguity occur in multiple inheritance?
3. If a base class and a derived class each include a member function with the same name, the member function of the derived class will be called by an object of the derived class. State true or false.
4. Is it illegal to make objects of one class as members of another class?

Review Questions [Objective Types]

5. How abstract class is related to pure virtual function?
6. Is it legal to create a pointer of an abstract base class?

BPIT

Review Questions [Short Answer Types]

1. When should one derive a class publicly or privately?
2. How does inheritance influence the working of constructors and destructors?
3. Class Y has been derived from class X. The class Y does not contain any data member of its own. Does the class Y require constructor? If yes, why?
4. What is containership? How does it different from inheritance?
5. Is constructor overloading different from ordinary function overloading? How? Can you overload a destructor?

Review Questions [Short Answer Types]

6. What does the term disambiguation suggest?
7. What are virtual base class? When should they be used?
8. What is object slicing? Give example from a C++ program
9. Is it necessary that the virtual function overridden in the derived class must have the same signature?
10. A function template have multiple argument types.. Discuss with example.

Review Questions [Long Answer Types]

1. What is visibility mode? What are the different visibility mode supported by C++?
2. What are the different form of inheritance? Explain with an example.
3. List the operators that can not be overloaded and justify why they can not be overloaded?
4. Write a program to overload unary operator, say ++ for incrementing distance in FPS system. Describe the working model of an overloaded operator with the same program.

Review Questions [Long Answer Types]

5. Explain the syntax of binary operator overloading. How many arguments are required in the definition of an overloaded binary operator?
6. Suggest and implement a program to trace memory leakage.
7. What is runtime dispatching? Explain with examples how C++ handle run time dispatching?
8. What are the virtual destructors? How do they differ from normal destructors? Can constructors be declared as virtual constructors? Give reasons.

Review Questions [Long Answer Types]

9. A function template can be overloaded. Write a program in C++ to support the view.
10. Can we distribute function template and class templates in object libraries?

Recommended Books

TEXT:

1. A..R.Venugopal, Rajkumar, T. Ravishanker "Mastering C++", TMH, 2009.
2. S. B. Lippman & J. Lajoie, "C++ Primer", 6th Edition, Addison Wesley, 2006.

REFERENCE:

1. R. Lafore, "Object Oriented Programming using C++", Galgotia Publications, 2008.
2. D . Parasons, "Object Oriented Programming with C++", BPB Publication.
3. Steven C. Lawlor, "The Art of Programming Computer Science with C++", Vikas Publication.
4. Schildt Herbert, "C++: The Complete Reference", 7th Ed., Tata McGraw Hill, 2008.