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 an 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.

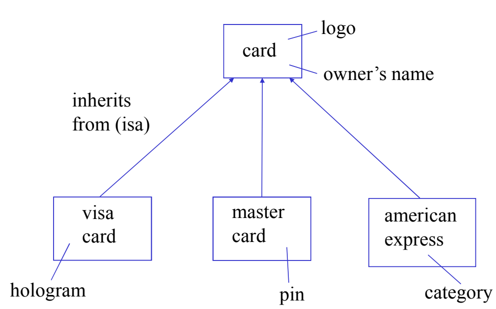
Inheritance Examples

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

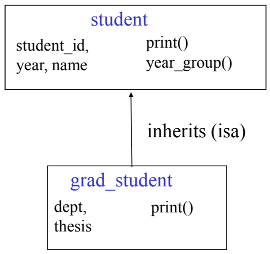
More Examples

Base Class	Derived Classes
Employee	Manager Researcher Worker
Account	CheckingAccount SavingAccount

Credit cards



The Student Class Hierarchy



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
}
```

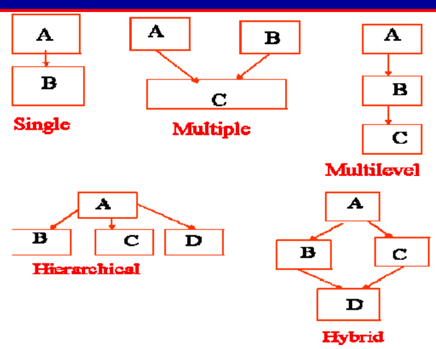

visibility mode

- 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
- Multipath 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

Single level Inheritance

```
#include<iostream.h>
class B
{
        private:
          int a;
        public:
          int b;            //publicly inherited
          void get_ab();
          int get_a(void);
          void show_a(void);
};
class D : public B
{
   private:                        // public derivation
         int c;
    public:
        void mul(void);
        void display(void);
};

   void B::get_ab()
      {
          a=5;  b=10;
      }
   int B::get_a()
      {  return a;}
   void B::show_a()
      {  cout<<"a = "<<a<<"\n";  }
   void D::mul()
      {  c=b*get_a();  }
   void D::display()
      {   cout<<"a = "<<get_a()<<"\n";
         cout<<"b = "<<b<<"\n";
         cout<<"c = "<<c<<"\n";
      }
void main()
{
   D d;d.get_ab(); d.mul(); d.show_a(); d.display(); d.b=20;   d.mul(); d.display();
}
```

```
(Inactive C:\TCWIN45\BIN\NONAME02.EXE)
a = 5
a = 5
b = 10
c = 50
a = 5
b = 20
c = 100
```


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();
};
```

Program of Single 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();
}
```


Program of Single 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;
}
 main()
{
clrscr();
student s1; // Create an object of student type
```


Program of Single Inheritance

```
s1.getdata();// Read data of a student
cout<<"\n The student data is…:;
cout<<"\n";
obj.dispaly_data();//inherit function from class person
obj.show_data();
return(0);
}
```

MultiLevel Inheritance

Turbo C++ - [c:\tcwin45\bin\noname02.cpp]

```
//To show the working of multilevel inheritance.

#include <iostream.h>
class student
{
   protected:
      int roll_number;
   public:
         void get_number(int);
         void put_number(void);
};
void student::get_number(int a)
{
   roll_number = a;
}
void student::put_number()
{
   cout<<"Roll number  "<<roll_number<<"\n";
}

class test : public student
{
   protected :
      float sub1;
      float sub2;
   public:
        void get_marks(float,float);
        void put_marks(void);
};
void test:: get_marks(float x,float y)
      {          sub1=x;    sub2=y;       }
void test::put_marks()
   {
      cout<<"Marks in sub1 "<<sub1<<"\n";
      cout<<"Marks in sub2 "<<sub2<<"\n";
   }
class result: public test
{
      float total;
```



```
   public:
         void display(void);
};
void result :: display(void)
{
   total=sub1+sub2;
   put_number();
   put_marks();
   cout<<"total = "<<total;
}
void main()
{
   result S1;
   S1.get_number(111);
   S1.get_marks(75.0,59.5);
   S1.display();
}
```

[Inactive C:\TCWIN45\BIN\NONAME02.EXE]

```
Roll number  111
Marks in sub1 75
Marks in sub2 59.5
total = 134.5
```

Program running

Multilevel Inheritance

```
#include<iostream.h>
#include<conio.h>
class A1  // Base class
{
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;
```

Multilevel Inheritance

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

OUTPUT:

Name : Amit
Age : 26 Years
Sex : M
Height : 6 Feets
Weight : 68 kg

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()
{
clrscr();
E   x;                  // Object declaration
x.getdata();        // Reads data
x.showdata();     //Displays data
}
```

OUTPUT:

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;
```


Example of Hierarchial Inheritance

```
public :
void get_data();
char compute_grade();
void show_data();
};

//Derive a subclass employee
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

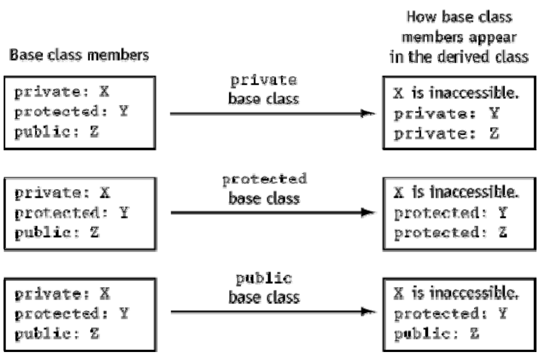
```
void person :: read_data()
{
cout<<"\n Enter name:"; cin>>name;
cout<<"\n Enter age: "; cin>>age;
}

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;
}
```

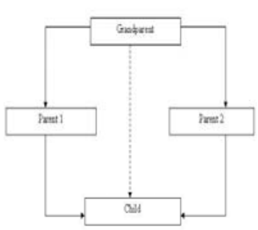
Example of Hierarchial Inheritance

```
main()
{
clrscr();
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…:;
cout<<"\n";
obj.dispaly_data();//inherit function from class person
obj.show_data();
e1.input_emp(); // Read Employee data
cout<<"\n The Employee data is…:;
cout<<"\n";
e1.dispay_data ();// Inherit function from class person to display
e1. disp_emp ();   // Display Employee details
return(0);
}
```

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

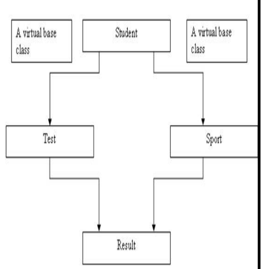
- Such inheritance by the child class may create some problems.
- **All the public and protected members of the grandparent class are inherited into the child class twice; first via parent1 class and then again via parent2 class.**
- **This means that the child class would have duplicate set of members 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

```
class grandparent
{
        -------
        -------
};

class parent1: virtual public grandparent
{
        -------
        -------
};

class parent2: virtual public grandparent
{
        -------
        -------
};

class child: public parent1, public parent2
{
        ------- // only one copy of grandparent class will be inherited
        -------
};
```


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.



Turbo C++ - [c:\tcwin45\bin\virtual.cpp]

File Edit Search View Project Debug Tool Options Window Help

```
// Program to show virtual base class
#include<iostream.h>
class student
{
   protected:
            int roll_number;
   public:
            void get_number(int a)
            {
               roll_number = a;
            }
            void put_number()
            {
               cout<<"\nThe Roll Number is   :  "<<roll_number;
            }
};
class test : virtual public student
{
   protected:
            float m1,m2;
   public:
            void get_marks(float x, float y)
            {
               m1=x;
               m2=y;
            }
            void put_marks()
            {
               cout<<"\nMarks Obtained :  ";
               cout<<"\n\t\t\tm1 =  "<<m1;
               cout<<"\n\t\t\tm2 =  "<<m2;
            }
};
class sports  :  public virtual student
{
   protected:
            int score;
   public:
            void get score(int a)
```



```
            {
               score = a;
            }
            void put_score()
            {
               cout<<"\n\nThe Score is :  "<<score;
            }
};
class result : public test, public sports
{
      float total;
   public:
      void display();
};
void result :: display()
      {
         total= m1 + m2 +score;
         put_number();
         put_marks();
         put_score();
         cout<<" \n\nTotal Score : "<<total;
      }
void main()
{
   result student1;
   student1.get_number(115);
   student1.get_marks(20.5, 25.5);
   student1.get_score(80);
   student1.display();
}
```

(Inactive C:\TCWIN45\BIN\VIRTUAL.EXE)

```
The Roll Number is      :       115
Marks Obtained  :
                        m1      =       20.5
                        m2      =       25.5

The Score is    :       80

Total Score     : 126
```

Program running

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(); // uses member of derived class
return 0;
}
```

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";}
};
```



```
// 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;
}
```


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 th 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"; }
};
```



```
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;
}
```


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.

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"; }
};
```

```
// 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();
}
};
```



```
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( ).**

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:
```

```
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;
}
```


Constructors and destructors in Base and derived Class

```
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// do nothing but construct and destruct ob
return 0;
}
```


Constructors and destructors in Base and derived Class

As the comment in **main( ) indicates, this program simply constructs and then** destroys an object called **ob that is of class derived.**

O/P-

Constructing base
Constructing derived
Destructing derived
Destructing base

Constructors and destructors in Base and derived Class

Constructors are called in order of derivation, destructors in reverse order.

```
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived1 : public base {
public:
derived1() { cout << "Constructing derived1\n"; }
~derived1() { cout << "Destructing derived1\n"; }
};
class derived2: public derived1 {
public:
derived2() { cout << "Constructing derived2\n"; }
~derived2() { cout << "Destructing derived2\n"; }
};
int main()
{
derived2 ob;
// construct and destruct ob
return 0;
}
```

displays this output:

Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base

Constructors and destructors in Base and derived Class

The same general rule applies in situations involving multiple base classes.For example-

```
class base1 {
public:
base1() { cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
public:
base2() { cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// construct and destruct ob
return 0;
}
```

produces this output:

Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1

Constructors and destructors in Base and derived Class

As you can see, constructors are called in order of derivation, left to right, as specified in **derived's inheritance list. Destructors are called in reverse order, right to left. This** means that had **base2 been specified before base1 in derived's list, as shown here:**

class derived: public base2, public base1 {

then the output of this program would have looked like this:

Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2

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**

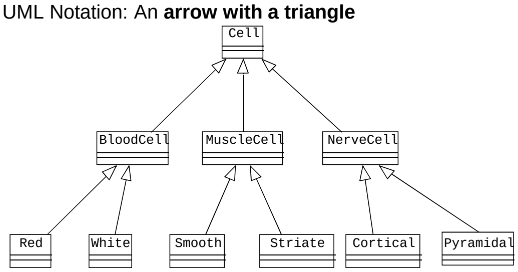
base::j;

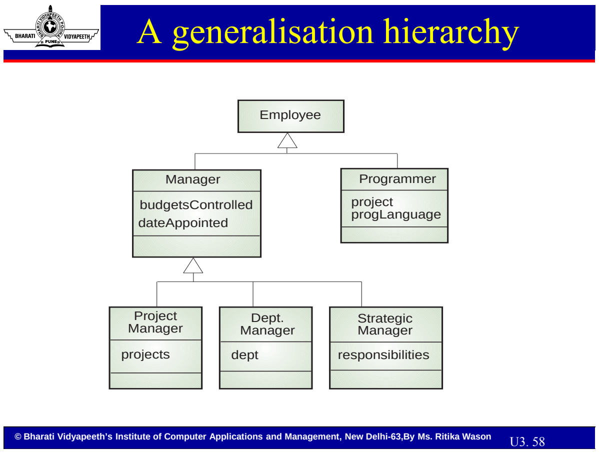
Generalisation and inheritance

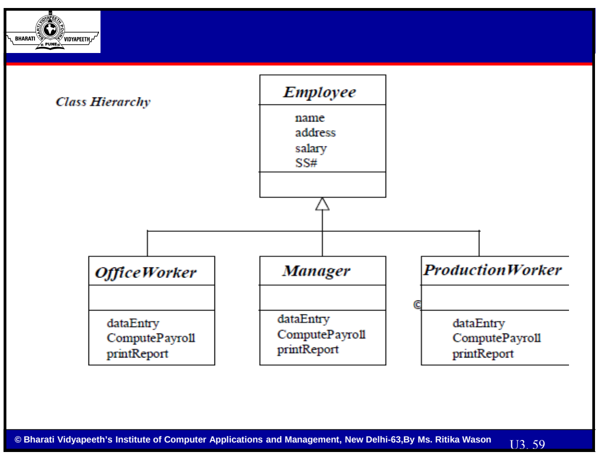
- Objects are members of classes which define attribute types and operations
- Classes may be arranged in a class hierarchy where **one class (a super-class) is a generalisation of one or more other classes (sub-classes)**
- A sub-class **inherits the attributes** and operations from its super class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

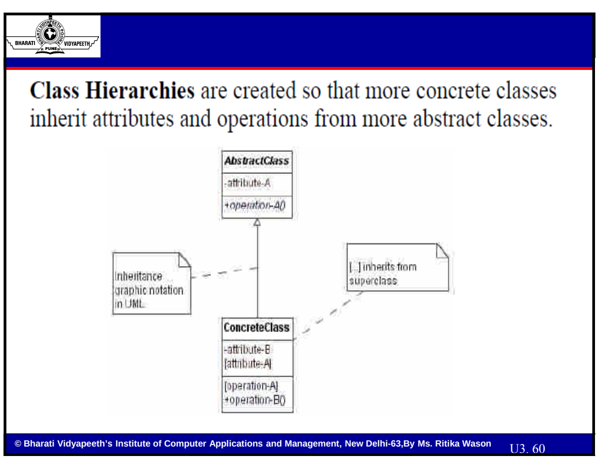
Inheritance

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



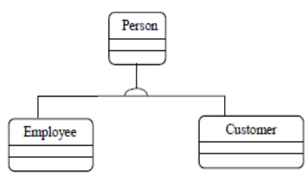

A generalisation hierarchy
Employee
Manager
budgetsControlled
dateAppointed
Programmer
project
progLanguage
Project Manager
projects
Dept. Manager
dept
Strategic Manager
responsibilities
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,By Ms. Ritika Wason
U3. 58


Class Hierarchy
Employee
name
address
salary
SS#
OfficeWorker
dataEntry
ComputePayroll
printReport
Manager
dataEntry
ComputePayroll
printReport
ProductionWorker
dataEntry
ComputePayroll
printReport
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,By Ms. Ritika Wason
U3. 59

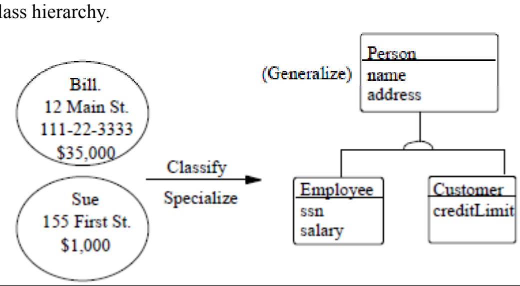

Class Hierarchies are created so that more concrete classes inherit attributes and operations from more abstract classes.
AbstractClass
-attribute-A
+operation-A()
inheritance graphic notation in UML
[...] inherits from superclass
ConcreteClass
-attribute-B
[attribute-A]
[operation-A]
+operation-B()
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63,By Ms. Ritika Wason
U3. 60

•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.**

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.

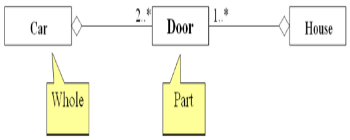


A mechanism for expressing similarity among classes. It represents generalization (What is the same?) and specialization (What is different?), making common attributes and services explicit within a class hierarchy.



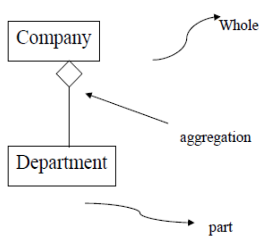
Aggregation or containership or object Composition in C++

A special form of association that models a whole-part relationship between a

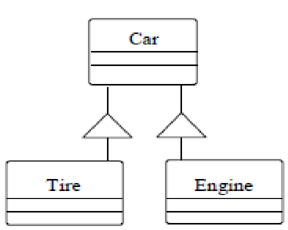


Aggregation is also called **whole-part or HasA.**
For example, **an Aircraft contains an Engine or in other words, an Aircraft has an Engine.**

Aggregation
- Represents a **"has-a" (whole-part) relationship**
- **An object of the whole has objects of the part**

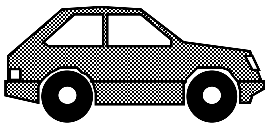


For example, among other things, **a car consists of tires** and an engine. Note that **the opposite of aggregation is decomposition.**

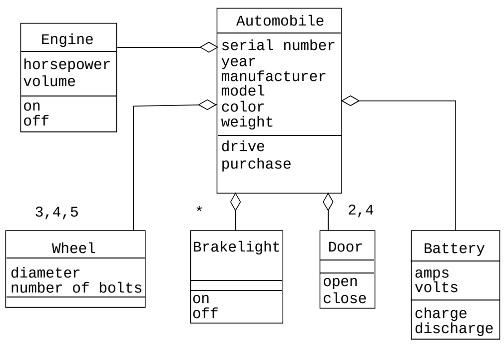


Aggregation

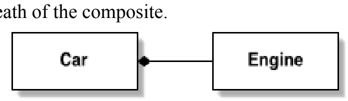
- Models "part of" hierarchy
- Useful for modeling the breakdown of a product into **its component** parts .
- UML notation: Like an association but with a small **diamond indicating** the assembly end of the relationship.



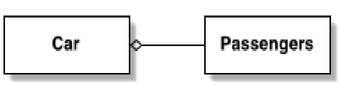
Aggregation



A form of aggregation with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it (i.e., they share lifetimes). Such parts can also be explicitly removed before the death of the composite.



Composition: every car has an engine.

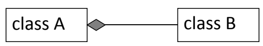


Aggregation: cars may have passengers, they come and go

•*Containment or contenership:* 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 represnt it in diagram like this.



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

```
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++).


```
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();
}
};
```



```
int main()
{
Employee obj;
obj.setEmployee();
obj.displayEmployee();
return 0;
}
```

```
class c{
public :
        c(){cout<<"c const \n";};
        ~c(){cout<< "c dest \n";};};
class a{
        c obj;
public :
        a(){cout<<"a const \n";};
        ~a(){cout<< "a dest \n";};
        void fn(){ cout<<"base\n";}};
```



```
void main()
{
        a obj;
}
```

RUN

c const
a const
a dest
c dest
Press any key to continue

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";}
```



```
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

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 ( ) ;

```
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();    }
```


Turbo C++ - [c:\tcwin45\bin\derived_class.cpp]

```
// program to show how constructors are invoked in derived class
#include<iostream.h>
class alpha
{
   private:
            int x;
   public:
            alpha(int i)
            {
               x=i;
               cout<<"\nalpha initialized\n";
            }
            void show_x()
            {
               cout<<"\nx = "<<x;
            }
};
class beta
{
   private:
            float y;
   public:
            beta(float j)
            {
               y=j;
               cout<<"\nbeta initialized\n";
            }
            void show_y()
            {
               cout<<"\ny = "<<y;
            }
};
class gamma : public beta, public alpha
{
   private:
            int m,n;
   public:
            gamma(int a,float b, int c, int d): alpha(a), beta(b)
            {
               m=c;
               n=d;
               cout<<"\ngamma initialized\n";
            }
            void show_mn()
            {
               cout<<"\nm = "<<m;
               cout<<"\nn = "<<n;
            }
};

void main()
{
   gamma g(5,7.65,30,100);
   cout<<"\n";
   g.show_x();
   g.show_y();
   g.show_mn();
}
```

(Inactive C:\TCWIN45\BIN\DERIVED_.EXE)

```
beta initialized

alpha initialized

gamma initialized


x = 5
y = 7.65
m = 30
n = 100
```

Abstract Classes

- An abstract class is one that has no instances and is not designed to create objects
- Only designed to be inherited
- Provides a frame work, upon which other classes can be built
- Normally exist at the root of the hierarchy

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.

Conclusion

- 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.

POLYMORPHISM

Learning Objectives

- **Polymorphism**
- **Categorization of Polymorphism techniques**
- **Function Overloading**
- **Operator Overloading**
- **Run type polymorphism/ Virtual Function**

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).

POLYMORPHISM (Contd.)

- 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.

FUNCTION OVERLOADING

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

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


Contd…

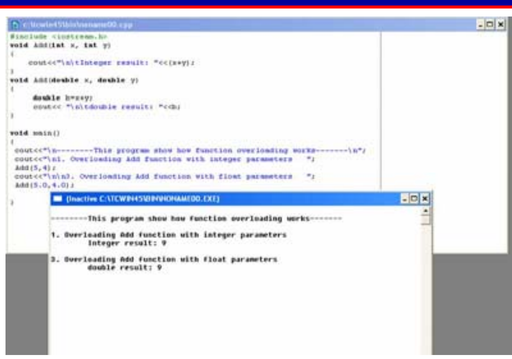
When a function, with same name, is declared more than once in the program, the compiler will interpret the second declaration as follows:

- 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
```


Contd…



OPERATOR OVERLOADING

Unary Operator Overloading

```
// Program1.cpp:  Index class with operator overloading for increment operator

 #include  < iostream.h>

class Index
{
   private:
     int  value;                     //  Index  Value
   public:
     Index ( )                       //  No argument constructor
       {
          value  =  0;
       }
      int GetIndex( )             // Index Access
        {  return value;
        }
      void operator + +( )      // prefix or postfix increment operator
         {
            value = value + 1;    // value+ + ;
         }
} ;
```


Unary Operator Overloading

```
void main( )
{  Index idx1, idx2;              // idex1 and idx2 are objects of  Index class
 // display index values
    cout << " \nIndex1 = " << idx1 . GetIndex ( ) ;
    cout << " \nIndex2 = " << idx2 . GetIndex ( ) ;

    // Advance Index objects with + + operators
   + + idx1;                // equivalent to idx1. operator + + ( ) ;
   idx2 + + ;
   idx2 + + ;
  cout  <<  " \ nIndex1 = " << idx1. GetIndex ( ) ;
  cout  <<  " \ nIndex2 = " << idx2. GetIndex ( ) ;
}
```

<u>RUN</u>

Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2

Operator Returns Values

The operator function in the previous program Program1.cpp has a subtle defect. An attempt to use an expression such as;

idx1= idx2++;

will lead to a compilation error like *Improper Assignment* because the return type of operator++ is defined as void type. Such an assignment operation can be permitted after modifying the return type of the operator ++() member function of the index class.

```
//Program2.cpp:  Index class with overloaded operator returning an object
#include  < iostream.h>
class Index
{
   private:
       int  value;                      //  Index  Value
  public:
       Index ( )                        //  No argument constructor
      {  value  =  0;
       }
      int GetIndex( )             // Index Access
       { return value;
        }
```


Operator Returns Values

```
Index operator + +( )      // Returns nameless object of class Index
        {     Index temp;                   // temp object
               value = value + 1;         // update index value
                temp.value = value;     //  Initialise temp object
               return temp;                //  Returns temp object
          }
} ;
 void main( )
{
      Index idx1, idx2;              // idex1 and idx2 are objects of  Index class
    // display index values
    cout << " \nIndex1 = " << idx1 . GetIndex ( ) ;
    cout << " \nIndex2 = " << idx2 . GetIndex ( ) ;
    // Returned object of  idx2+ +  is assigned to object idx1
   idx1 = idx2++; // invokes the overloaded function and assigned the return value to the object  idx1 of the
      index
   idx2 + + ;   // Returned object of idx2++ is unused
   cout  <<  " \ nIndex1 = " << idx1. GetIndex ( ) ;
  cout  <<  " \ nIndex2 = " << idx2. GetIndex ( ) ;
}
```


Operator Returns Values

RUN

Index1 = 0
Index2 = 0
Index1 = 1
Index2 = 2

Binary Operator Overloading

Complex numbers consists of two parts: real part and imaginary part. It is represented as (x+iy), where x is the real part and y is the imaginary part. The process of performing the addition operation is illustrated below. Let c1, c2and c3 be three complex numbers represented as follows:

c1 = x1 + i y1;
c2 = x2 + i y2;

The operation c3 = c1 + c2 is given by

c3 = (c1 .x1 + c2 .x2) + i (c1. y1 + c2. y2) ;

```
// Complex1.cpp: Complex numbers operations with binary operator overloading
# include < iostream.h>
class complex
{
  private :
      float real ;        // real part of complex number
      float imag ;        // imaginary part of complex number
  public :
      complex ( )                  // no argument constructor
      {
          real = imag = 0 . 0 ;
      }
```


Binary Operator Overloading

```
   void getdata ( )            // read complex number
      {     cout << "Real Part ? " ;
            cin >> real ;
            cout << "Imag Part ? " ;
            cin >> imag ;
      }
    complex operator + ( complex c2 ) ;   // complex addition
    void outdata ( char *msg )          // display complex number
    {     cout << endl << msg ;
          cout << " ( " << real ;
          cout << " , " << imag<<" ) " ;
    }
} ;
 // add default and c2 complex objects
complex complex : : operator + ( complex c2 )
{ complex temp ;                       // object temp of complex class
   temp . real = real + c2 . real ;    //add real parts
   temp . imag = imag = c2 . imag ;    //add imaginary parts
   return ( temp ) ;                   // return complex object
}
```


Binary Operator Overloading

```
void main ( )
{
      complex c1, c2 , c3 ;      // c1 , c2 , c3 are object of complex class
      cout << " Enter Complex Number c1 . . " << endl ;
      cl . getdata ( ) ;
      cout << " Enter Complex Number c2 . . " << endl ;
      c2 . getdata ( ) ;
      c3 = c1 + c2 ; // add c1 and c2 and assign the result to c3 i.e; c3 = c1 . operator+ ( c2 ) ;
      c3 . outdata ( " c3 = c1 + c2 : " ) ; // display result
}
```

<u>RUN</u>

Enter Complex Number c1 . .
Real Part ? <u>2.5</u>
Imag Part ? <u>2.0</u>
Enter Complex Number c2 . .
Real Part ? <u>3.0</u>
Imag Part ? <u>1.5</u>
C3 = c1 + c2 : (5 . 5 , 3.5)

Runtime Polymorphism

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.

Need For Virtual Functions

When objects of different class in a class hierarchy, react to the same message in their own unique ways, they are said to exhibit polymorphic behaviour. This program has the base class Father and the derived class Son and has a member function show() with the same name and prototype. In C++, a pointer to the base class can be used to point to its derived class objects.

```
//Parent1.cpp: Invoking DC members through BC pointer
#include<iostream.h>
#include<string.h>
 class Father   // Father's name
{ char name[20];
  public:
    Father(char *fname)
     {strcpy(name, fname);  //fname contains Father's name
     }
    void show() //show in base class
     {cout<<"Father's name:" << name << endl;
     }
};
class son: public Father
{ char name[20];  // Son's name
  public:
// 2 Argument constructor; invokes 1 argument constructor of Father
```

Need For Virtual Functions

```
Son( char *sname, char *fname) : Father(fname)
  { strcpy(name, sname);  // sname contains Son's name
  }
 void show() //show in derived class
  { cout<<"Son's name:" << name << endl;
  }
};
 void main()
{     Father *fp;     // pointer to Father's class object
      Father f1 ( " RAVI");
     fp = &f1;       // fp points to Father class object
     fp->show();   // Display Father show() function
     Son s1 ( " AMAN", "RAVI");
     fp = &s1;       // valid assignment
     fp->show();   // guess what is output? Father or Son!
}
```

OUTPUT:

Father's name: RAVI

Father's name: RAVI

Need For Virtual Functions

- It is interesting to note that after **valid assignment to the address of object s1 of the class Son to fp, it still invokes the member function show() defined in the class Father.**
- **Hence we need Runtime polymorphism (i.e. Virtual Function) that allows to postpone the decision of selecting the suitable member functions until runtime.**
-
- **In this case, a member function to be invoked depend on the class's object to which the pointer is pointing.**

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

};

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;
}
```


- 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:


```
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();
}
```

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.

•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.***


```
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,**

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() {};
};
```


•**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.

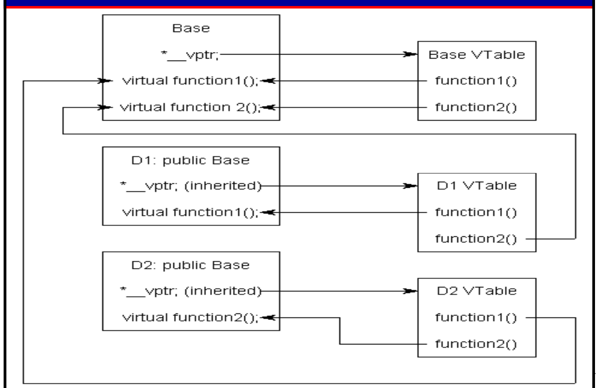
•Now, let's talk about how these virtual tables are filled out. Because **there are only two virtual functions here, each virtual table will have two entries (one for function1(), and one for function2()).** Remember that when these virtual tables are filled out, each entry is filled out with the most-derived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

•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:



the *__vptr in each class points to the virtual table for that class. The entries in the virtual table point to the most-derived version of the function objects of that class are allowed to call.

So consider what happens when we create an object of type D1:

```
int main()
{
   D1 cClass;
}
```

Because cClass is a D1 object, cClass has it's *__vptr set to the D1 virtual table.

Now, **let's set a base pointer to D1:**

```
int main()
{
   D1 cClass;
   Base *pClass = &cClass;
}
```

So what happens when we try to call pClass->function1()?

```
int main()
{
   D1 cClass;
   Base *pClass = &cClass;
   pClass->function1();
}
```


•First, the program recognizes that function1() is a virtual function. Second, uses pClass->__vptr to get to D1′s virtual table. Third, it looks up which version of function1() to call in D1′s virtual table. This has **been set to D1::function1(). Therefore, pClass->function1() resolves to D1::function1()!**

•Now, you might be saying, "But what if Base really pointed to a Base object instead of a D1 object. Would it still call D1::function1()?". The answer is no.

```
int main()
{
   Base cClass;
   Base *pClass = &cClass;
   pClass->function1();
}
```


In this case, when cClass is created, __vptr points to Base's virtual table, not D1′s virtual table. Consequently, **pClass->__vptr will also be pointing to Base's virtual table. Base's virtual table entry for function1() points to Base::function1().** Thus, pClass->function1() resolves to Base::function1(), which is the most-derived version of function1() that a Base object should be able to call.

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;
```


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

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

- 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.

Working with Files

Learning Objectives

- **Console User Interaction**
- **Input Output Stream**
- **File Stream Classes**
- **Opening a file with open()**
- **Opening a file with constructors**
- **End-of- file detection**
- **File modes**
- **File pointers**

Learning Objectives

- Sequential File Operation
- Random Access Files
- Error Handling
- Command-line Arguments

Input/Output with Files

• C++ has support both for input and output with files through the following classes:

• ofstream: File class for writing operations (derived from ostream).
• ifstream: File class for reading operations (derived from istream).
• fstream: File class for both reading and writing operations (derived from iostream).

Open a file

• First operation generally done on an object of one of these classes is to associate it to a real file.

• In order to open a file with a stream object, we use its member function **open():**
void open (const char * *filename*, **openmode** *mode*);

• where *filename* is a string of characters representing the name of the file to be opened.

Open a file Cont..

•*Mode* **is a combination of the following flags:**

ios::in Open file for reading
ios::out Open file for writing
ios::ate Initial position: end of file
ios::app Every output is appended at the end of file.
ios::trunc If the file already existed it is erased.
ios::binary Binary mode

•**Flags can be combined using bitwise operator OR: | .**

```
ofstreamfile;
file.open ("example.bin", ios::out | ios::app | ios::binary);
```


Open a file (Cont..)

• Member functions **open** of classes **ofstream, ifstream** and **fstream** include a default mode.

Class	**Default *Mode* to Parameter**
Ofstream	ios::out | ios::trunc
Ifstream	ios::in
Fstream	ios::in | ios::out

Open a file (Cont..)

- Include a constructor that directly calls the **open** member functions and has the same parameters as
- ofstream file("example.bin", ios::out| ios::app| ios::binary
- Check if a file has been correctly opened by calling the member function **is_open() :**
- **bool is_open();**
- Indicating **true** in case that indeed the object has been correctly associated with an open file or **false** otherwise.

Closing a File

- Call the member function **close().**
- Has the function of flushing the buffers and closing the file. Its form is quite simple:

  void close();
- Once this member function is called, the stream object can be used to open another file and the file is available again to be opened by other processes.
- In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close.

Text Mode Files

• Use the same members of these classes that we used in communication with the console.

// writing on a text file

```
#include <fstream.h>
int main () {
ofstream examplefile ("example.txt");
if (examplefile.is_open())
{examplefile << "This is a line.\n";
examplefile << "This is another line.\n";
examplefile.close();
}
return 0;}
```


eof() Inherits from Class ios

// Reading a text file

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib .h>
int main () {
char buffer[256];
ifstream examplefile ("example.txt");
if (! examplefile.is_open())
{ cout << "Error opening file";
exit (1); }
while (! examplefile.eof () )
{ examplefile.getline (buffer,100);
cout << buffer << endl; }
return 0;
}
```

get and *put* Stream Pointers

All i/o streams objects have, at least, one stream pointer:
• **ifstream,** has a pointer known as ***get pointer*** that points to the next element to be read.
• **ofstream,** has a pointer ***put pointer*** that points to the location where the next element has to be written.
• Finally **fstream,** inherits **both: *get* and *put.***

• These stream pointers that point to the reading or writing locations within a stream can be read and/or manipulated using the following member functions:

Functions for get & put pointers

• **tellg() and tellp()**
These two member functions admit no parameters and return the current position of *get* stream pointer (in case of tellg) or *put* stream pointer (in case of tellp).

• **seekg() andseekp()**
1. To change the position of stream pointers *get* and *put*. Both functions are overloaded with two different prototypes:
seekg (pos_type *position*);
seekp (pos_type *position*);
The stream pointer is changed to an absolute position from the beginning of the file.

Functions for get & put pointers Cont..

• **seekg (off_type** *offse***t, seekdir** *direction* **);**
seekp(off_type *offse***t, seekdir** *direction* **);**

• Using this prototype, an offset from a concrete point determined by parameter *direction* can be specified. It can be:
ios::beg offset specified from the beginning of the stream.
ios::cur offset specified from the current position of the stream pointer.
ios::end offset specified from the end of the stream.

Obtaining File Size

```
#include <iostream.h>
#include <fstream.h>
const char * filename = "example.txt";
int main () {
long l,m;
ifstream file (filename, ios::in | ios::binary);
l = file.tellg();
file.seekg (0, ios::end);
m = file.tellg();
file.close();
cout << "size of " << filename;
cout << " is " << (m-l) << " bytes.\n";
return 0; }
```


Binary Files

•Use **read** & **write** functions of **istream** and **ostream** class respectively for input output.

•Object of class **fstream** have both. Their prototypes are:

write (char *buffer, **streamsize** size);

read (char *buffer, **streamsize** size);

•**Buffer** is the address of a memory block where the read data are stored or from where the data to be written are taken. The **size** parameter is an integer value that specifies the number of characters to be read/written from/to the **buffer**.

Reading Writing Binary File

```
const char *filename="example.txt";
int main() {
char *buffer;
long size;
ifstream file (filename, ios::in | ios::binary | ios::ate);
size = file.tellg();
file.seekg(0, ios::beg);
buffer = new char [size];
```

Reading Writing Binary File

```
file.read (buffer, size);
file.close();
cout<<"the complete file is in buffer";
ofstream file1 ("test1.txt", ios::out | ios :: binary);
file1.write (buffer, size);
delete[] buffer;
return 0; }
```


get() and *put()*

- To read and write data

```
istream& get(char &ch)
ostream& put(char& ch)
while (in.get(ch))
    cout<<ch;
for(i=0; i<256;i++)
out.put((char)i);
```


Conclusion

- The C++ I/O system contain classes such as ifstream, ofstream and fstream to deal with file handling.
- These classes are derived from fstreambase class and are declared in a header file iostream.
- The fstream class does not provide a mode by default and therefore we must provide the mode explicitly.
- The class ios supports many member functions for managing many errors during file operations.

Exceptional Handling

Learning Objectives

- Errors and Exceptions
- Throwing mechanisms
- Multiple Catching
- Rethrowing exceptions
- Exception handling mechanism
- Catching all exceptions
- Restricting exceptions thrown

Exception Handling

- Exceptions are of two kinds, namely, synchronous exceptions and asynchronous exceptions.
- Errors such as "out-of-range index" and "over-flow" belong to the synchronous type exceptions.
- The errors that are caused by events beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions.
- The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

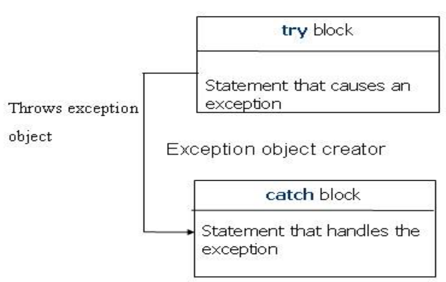
Exception Handling

- The purpose of the exception handling mechanism is to provide means to detect and report an exceptional circumstance or condition, so that appropriate action can be taken against it. The mechanism suggests a separate error handling code that performs the following tasks:
- Find the problem (**Hit** on the exception).
- Inform that an error has occurred (**Throw** the exception).
- Receive the error information (**Catch** the exception).
- Take corrective actions (**Handle** the exception).
- The error handling code basically consists of two segments; one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate action.

Exception Handling Mechanisms

- C++ exception handling mechanism is basically built upon three keywords, namely, **try** , **throw**, and **catch** .
- The keyword **try** is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements is known as try block.
- When an exception is detected, it is thrown using a **throw** statement in the try block.
- Catch blocks defined by the keyword **catch,** 'catches' the exception 'thrown' by the **throw** statement in the try block, and handles it appropriately.

Exception Handling Mechanisms



The catch block that catches an exception must immediately follow the try block that throws the exception.

The general form of these two blocks is as follows:

```
try
{
        -------
        throw exception;
        -------
}
catch(exception-type object)
{
        -------
        -------
}
```

1. When a try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block. Note that exceptions are objects used to transmit information about a problem.
2. If the type of object thrown matches the argument- type in the catch statement, then catch block is executed for handling the exception.
3. If it does not match, the program is aborted with the help of the abort() function which is invoked by default.
4. When no exception is detected and thrown, the control goes to the statement immediately after the catch block, i.e., the catch block is skipped.


```
#include<iostream.h>
void main()
{
    int a,b;
    cout<<"Enter the value of a and b\n";
    cin>>a;
    cout<<"\n";
    cin>>b;
    int x=a-b;
    try
    {
        if(x!=0)
        {
            cout<<"Result(a/x) ="<<a/x;
        }
        else
        {
            throw(x);
        }
    }
    catch(int i)
    {
        cout<<"\nException caught : x = "<<x;
    }
    cout<<"\nEND";
}
```

```
(Inactive C:\TCWIN45\BIN\THIS.EXE)
Enter the value of a and b
3

3

Exception caught : x = 0
END
```

The above program detects and catches a division-by-zero error.

Different values of a and b will show how the exception mechanism works.

Throw Mechanism

- When an exception, which is desired to be handled, is detected, it is thrown using the **throw** statement. It can be used in any one of the following form:
- throw(exception) ;
- throw exception ;
- throw ; // used for re-throwing an exception

Throw Mechanism

- The object passed to the **throw** statement may be of any type, including constants.
- It is also possible to throw objects not intended for error handling. When an exception is thrown, it will be caught by the **catch** statement associated with the try block, i.e., the control exits the current try block, and is transferred to the catch block.
- The throw point can be in a deeply nested scope within a try block or in a deeply nested function call. In any case, control is transferred to the catch statement.

Catch Mechanism

- The code for handling exceptions is included in a catch block. A catch block looks like a function definition and is of the following form:

```
catch(type argument)
{
    // Statements for handling exceptions
```

- The **type** indicates the type of exception that a catch block handles. The parameter **argument** is the parameter's name. Note that, the exception-handling code is placed between the two braces.
- The **catch** statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed.

Catch Mechanism

1. When it is caught, the code in the catch block is executed.
2. If the parameter in the catch definition is given a name, then the parameter can be used in the exception handling code. After executing the handler, the control goes to the statement immediately following the catch block.
3. Due to a mismatch, if an exception is not caught, an abnormal program termination may occur. It is important to note that the catch block is simply skipped if the catch statement does not catch the exception.

Multiple Catch Statements

- Sometimes, it happens that a program segment has more than one condition to throw an exception.
- In such a situation, we can associate more than one catch statement with a try, as shown here :

```
try
{
        // try block;
}
catch(type-1 argument-1)
{
        // catch block-1
}
catch(type-2 argument-2)
{
        // catch block-2
}
catch(type-N argument-N)
{
        // catch block-N;
}
```


Multiple Catch Statements

- When an exception is thrown, the exception handlers are searched in order for an appropriate match.
- The first handler that yields a match is executed.
- After executing the handler, the control goes to the first statement after the last catch block for that try (i.e., all other handlers are bypassed).
- When no match is found, the program is terminated.


```
#include<iostream.h>

void test(int x)
{
        try
        {
                if(x==1)
                        throw x;
                else if(x==0)
                        throw 'x';
                else if(x==-1)
                        throw 1.0;
                cout<<"\n End of try-block\n";
        }
        catch(char c) //catch 1
        {
                cout<<"Caught a character \n";
        }
        catch(int m) //catch 2
        {
                cout<<"Caught an integer \n";
        }
        catch(double d) //catch 3
        {
                cout<<"Caught a double \n";
        }

        cout<<"\n End of try-catch block\n";
}
```

(Inactive C:\TCWIN45\BIN\MULCATCH.EXE)

```
Testing multiple catches
x == 1
Caught an integer

 End of try-catch block
x == 0
Caught a character

 End of try-catch block
x == -1
Caught a double

 End of try-catch block
x == 2

 End of try-block

 End of try-catch block
```

Multiple Catch Execution

- The program when executed first, invokes the function test() with x = 1 and therefore throws x as an int exception. This matches the type of the parameter m in catch2 and therefore catch2 handler is executed.
- Immediately after the execution, the function test() is again invoked with x = 0. This time, the function throws 'x', a character type exception and therefore the first handler is executed.
- Finally, the handler catch3 is executed when a double type exception is thrown. Note that every time only the handler which catches the exception is executed and all other handlers are bypassed.
- When the try block does not throw any exception and when it completes normal execution, the control passes to the first statement after the last catch handler associated with that try bock.

Re-throwing an Exception

- A handler may decide to re-throw an exception caught without processing it. In such situations, we may simply invoke throw without any arguments as shown below:

  throw;
- This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block. The following program demonstrates how an exception is re-thrown and caught.

Program to illustrate re-throwing of exception

```
#include<iostream.h>

void divide(int x, double y)
{
	cout<<"Inside function \n";
	try
	{
		if(y == 0.0)
			throw y;
		else
		cout<<"Division = "<<x/y<<"\n";
	}
	catch(double)
	{
		cout<<"Caught double inside function\n";
		throw;
	}
	cout<<" END OF FUNCTION \n";
}

void main()
{
	cout<<"Inside main \n";
```


Program to illustrate re-throwing of exception

(Inactive C:\TCWIN45\BIN\RETHROW.EXE)

```
Inside main
Inside function
Division = 5
 END OF FUNCTION
Inside function
Caught double inside funtion
Caught doubling inside main
End of main
```


Re-throwing an Exception

- When an exception is re-thrown, it will not be caught by the same catch statement or any other catch in that group. Rather, it will be caught by an appropriate catch in the outer try/ catch sequence only.
- A catch handler itself may detect and throw an exception. Here again, the exception thrown will not be caught by any catch statements in that group. It will be passed on to the next outer try/catch sequence for processing.

Exception Specification

- It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to the function definition. The general form of using an exception specification is:

```
return-type function-name(argument-list) throw (type-list)
{
        // function Body
}
```


Exception Specification

- The type-list specifies the type of exceptions that may be thrown by the function.
- Throwing any other type of exception will cause abnormal program termination.
- If we wish to prevent a function from throwing any exception, we may do so by making the type-list empty. That is, we must use,

```
throw(); // empty list
```


Conclusion

- Exceptions are of two types: synchronous and asynchronous. C++ provides mechanism for handling synchronous exceptions.
- An exception is typically caused by a faulty statement in a try block. The statement discovers the error and throws it which is caught by a catch statement.
- When an exception is not caught, the program is aborted.
- It is also possible to make a catch statement to catch all types of exceptions using ellipses as its arguments.

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?

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?

Research Problem

- Draw **Class Diagram and Generate Code** for **Health Care Center**. Write **C++ code for the same.**
- Patient Can arrange and cancel appointment with physician using scheduler. Physician decides to Prescribe Medication for Patient. Physician Specifies Drug Info: Medication Name, Dosage Amount, Number Doses & Refills. Computer Cross-Checks for Conflict Between Medication and Current Medications/Medical History Prescription Forwarded Electronically to Pharmacy or Else Printed for Patient .

Research Problem

- Draw **class diagram** with **code generation** using forward engineering for personal investment management system (PIMS) given below and write C++ Code for the same.
- Many people invest their money in a number of securities (shares). Generally, an investor has multiple portfolios of investments, each portfolio having investments in many securities. From time to time an investor sells or buys some securities and gets dividends for the securities. There is a current value of each security-many sites give this current value. It is proposed to build a personal investment management system (PIMS) to help investors keep track of their investments as well as on the overall portfolios. The system should also allow an investor to determine the net-worth of the portfolios.

Research Problem

- Consider an Online Airline Reservation System. You may want to check airline web sites to give you idea.
- Identify actors for Online Airline Reservation System. Explain the relevance of each actor.
- One use case is to make a Flight Reservation. List four additional use cases at a comparable level of abstraction. Describe each use case with exceptional flow.
- Draw **Class Diagram**. Elaborate **ticket class and Generate Code** for the same. Write C++ code also.

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.