UNIT IV

Inheritance: Introduction to inheritance, single inheritance, multi-level inheritance, multiple inheritance, hierarchical inheritance, hybrid inheritance.

Operator Overloading: Rules for overloading operators, overloading unary operators, overloading binary operators.

Pointers: Introduction to pointers, declaring and initializing pointers, arithmetic operations on pointers, pointers with arrays, arrays of pointers, pointers to objects, 'this' pointer.

4.1 INHERITANCE

4.1.1 Introduction to Inheritance

Q1. What is inheritance and explain it with syntax and benefits.

Ans :

The process of obtaining the data members and methods from one class to another class is known as **inheritance**. It is one of the fundamental features of object-oriented programming.

Important points

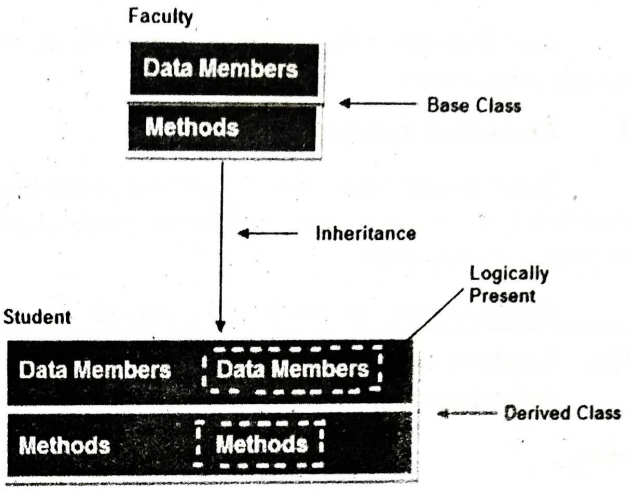
- In the inheritance the class which is give data members and methods is known as base or super or parent class.
- The class which is taking the data members and methods is known as sub or derived or child class.

Syntax

```
class subclass_name : superclass_name
{
    // data members
    // methods
}
```

Real life example of inheritance

The real life example of inheritance is child and parents, all the properties of father are inherited by his son.



In the above diagram data members and methods are represented in broken line are inherited from faculty class and they are visible in student class logically.

Advantage of inheritance

If we develop any application using this concept than that application have following advantages,

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

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

1. Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

class Subclass: public Superclass

2. Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass // By default its private inheritance

3. Protected Inheritance

In protected mode, the public and protected members of Super class becomes protected members of Sub class.

class subclass : **protected** Superclass

Q2. Explain briefly about various types of Inheritance.

Ans :

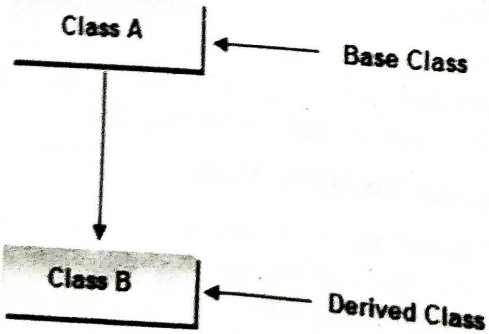
Types of Inheritance

Based on number of ways inheriting the feature of base class into derived class it have five types they are:

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

1. Single inheritance

In single inheritance there exists single base class and single derived class.



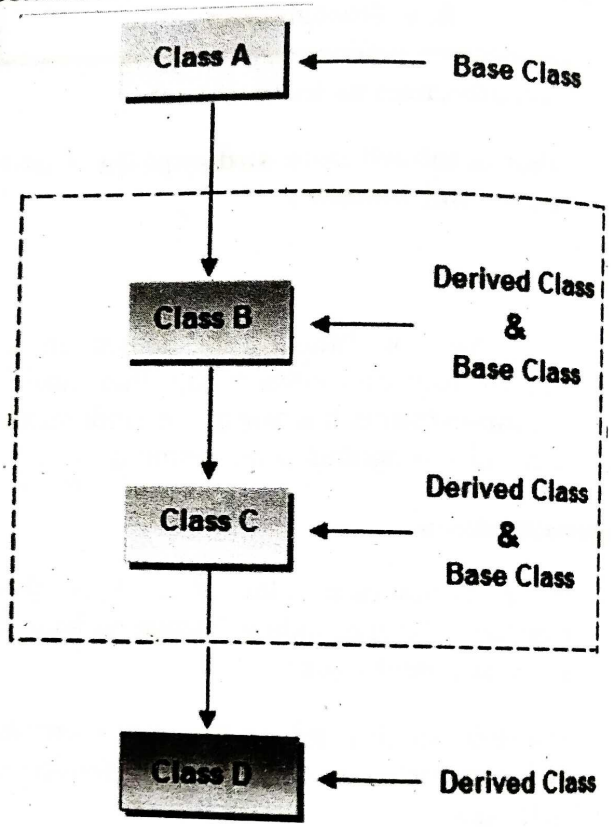
2. Multi level inheritances

In multi level inheritance there exists single base class, single derived class and multiple intermediate base classes.

Single base class + single derived class + multiple intermediate base classes.

Intermediate base classes

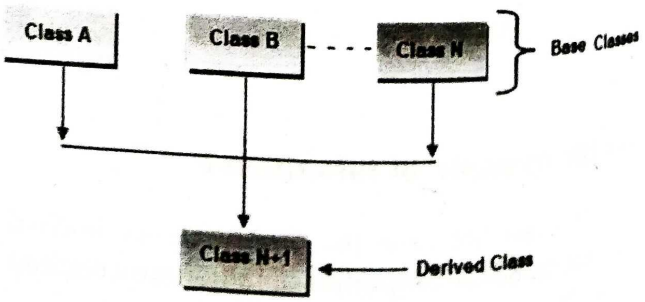
An intermediate base class is one in one context with access derived class and in another context same class access base class.



Hence all the above three inheritance types are supported by both classes and interfaces.

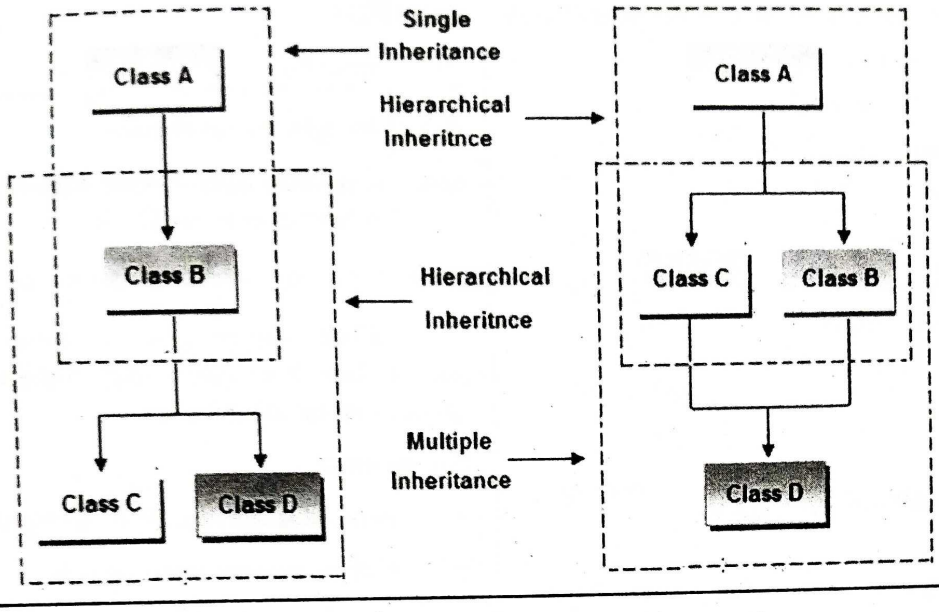
3. Multiple inheritance

In multiple inheritance there exist multiple classes and single derived class.



4. **Hybrid inheritance**

Combination of any inheritance type:



Q3. Explain how to define derived class in inheritance.

Ans :

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

How to Define Derive Class in Inheritance

First Define a base class as shown below:

```
class Base_Class_Name
{
}
```

Once the base class is defined you can define a derived class using following syntax:

```
class Derived_Class_Name: Visibility_Mode Base_Class_Name
{
}
```

Here,

Derived_Class_Name is the name of the class that you want to derive.

Visibility_Mode indicates the mode in which you want to derive a new class. It can be public, private or protected.

Base_Class_Name is the name of the class from which you want to derive a new class.

In the above syntax visibility mode determines how the data members of the base class are inherited in to derived class. Visibility mode can be public, private or protected. If you don't specify visibility mode then by default it is private.

Consider a base class Shape and its derived class Rectangle as follows:

```
// demonstration of base and derived classes
#include<iostream>
usingnamespace std;
// Base class
classShape
{
public:
void setWidth(int w)
{
width = w;
}
void setHeight(int h)
{
height = h;
}
protected:
int width;
int height;
};
// Derived class
classRectangle:publicShape
{
public:
int getArea()
{
return(width * height);
}
};
int main(void)
{
RectangleRect;
Rect.setWidth(5);
Rect.setHeight(7);
// Print the area of the object.
cout <<"Total area: "<<Rect.getArea()<<
endl;
return0;
}
```

When the above code is compiled and executed, it produces the following result:

OUTPUT

Total **area:35**

4.1.2 Single Inheritance

Q4. Explain about the concept of single inheritance on C++.

Ans : **(Imp.)**

When a single class is derived from a single parent class, it is called Single inheritance. It is the simplest of all inheritance.

For example,

- Animal is derived from living things
- Car is derived from vehicle
- Typist is derived from staff

Syntax

```
class base_classname
{
    properties;
    methods;
};
class derived_classname : visibility_mode
base_classname
{
    properties;
    methods;
};
```

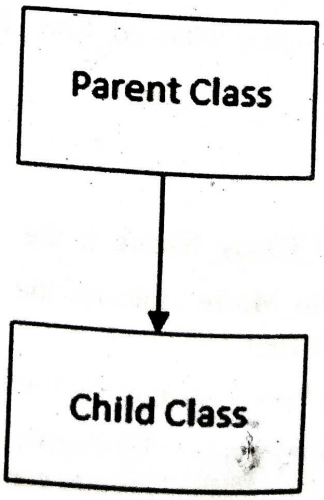


Fig.: Single Inheritance

Ambiguity in Single Inheritance in C++

If parent and child classes have same named method, parent name and scope resolution operator(::) is used. This is done to distinguish the method of child and parent class since both have same name.

Example of Single Inheritance in C++

//C++ Program to Inherit a Student class from Person Class printing the properties of the Student

```
#include <iostream>
#include <conio.h>
using namespace std;
class person    /*Parent class*/
{
  private:
     char fname[100],lname[100],gender[10];
  protected:
     int age;
  public:
     void input_person();
     void display_person();
};
class student: public person    /*Child class*/
{
   private:
      char college_name[100];
      char level[20];
   public:
      void input_student();
      void display_student();
};
void person::input_person()
{
    cout<<"First Name: ";
    cin>>fname;
    cout<<"Last Name: ";
    cin>>lname;
    cout<<"Gender: ";
    cin>>gender;
    cout<<"Age: ";
    cin>>age;
}
void person::display_person()
{
    cout<<"First Name : "<<fname<<endl;
    cout<<"Last Name : "<<lname<<endl;
    cout<<"Gender : "<<gender<<endl;
    cout<<"Age : "<<age<<endl;
}
void student::input_student()
{
    person::input_person();
    cout<<"College: ";
    fflush(stdin);
    gets(college_name);
    cout<<"Level: ";
    cin>>level;
}
void student::display_student()
{
    person::display_person();
    cout<<"College   : "<<college_name<<endl;
    cout<<"Level     : "<<level<<endl;
}
int main()
{
    student s;
    cout<<"Input data"<<endl;
    s.input_student();
    cout<<endl<<"Display data"<<endl;
    s.display_student();
    getch();
    return 0;
}
```

Output

Input data
First Name: Harry
Last Name: Potter
Gender: Male
Age: 23
College: Abc International College
Level: Bachelors
Display data
First Name : Harry
Last Name : Potter
Gender : Male
Age : 23
College : Abc International College
Level : Bachelors

4.1.3 Multi-Level Inheritance

Q5. Explain about multi – level Inheritance with example.

Ans : **(Imp.)**

When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent classes, such inheritance is called Multilevel Inheritance. The level of inheritance can be extended to any number of level depending upon the relation. Multilevel inheritance is similar to relation between grandfather, father and child.

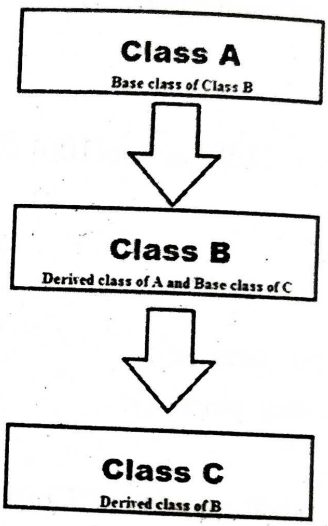
For example

- Student is derived from person and person is derived from class living things.
- Car is derived from vehicle and vehicle is derived from machine.

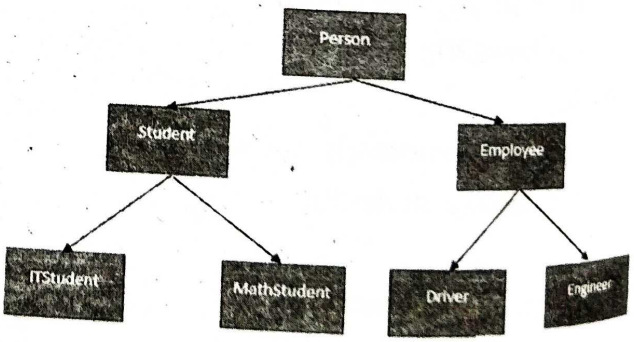
Syntax :

```
class base_classname
{
    properties;
    methods;
};
```

```
class intermediate_classname:visibility_mode base_classname
{
    properties;
    methods;
};
class child_classname:visibility_mode intermediate_classname
{
    properties;
    methods;
};
```



Below Image shows the example of multilevel inheritance



As you can see, Class Person is the base class of both Student and Employee classes. At the same time.

Class Student is the base for class IT Student and MathStudent classes. Employee is the base class for Driver and Engineer classes.

C++ program to create a programmer derived from employee which is himself derived from person using Multilevel Inheritance

```
#include <iostream>
#include <conio.h>
using namespace std;
class person
{
  char name[100],gender[10];
  int age;
  public:
    void getdata()
    {
       cout<<"Name: ";
       fflush(stdin);  /*clears input stream*/
       gets(name);
       cout<<"Age: ";
       cin>>age;
       cout<<"Gender: ";
       cin>>gender;
    }
    void display()
    {
       cout<<"Name: "<<name<<endl;
       cout<<"Age: "<<age<<endl;
       cout<<"Gender: "<<gender<<endl;
    }
};
class employee: public person
{
  char company[100];
  float salary;
  public:
    void getdata()
    {
       person::getdata();
       cout<<"Name of Company: ";
       fflush(stdin);
       gets(company);
      cout<<"Salary: Rs.";
      cin>>salary;
    }
    void display()
    {
       person::display();
            cout<<"Name of Company: "<<company<<endl;
       cout<<"Salary: Rs."<<salary<<endl;
    }
};

class programmer: public employee
{
   int number;
   public:
      void getdata()
      {
         employee::getdata();
            cout<<"Number of programming language known: ";
         cin>>number;
      }
      void display()
      {
         employee::display();
         cout<<"Number of programming
         language known: "<<number;
      }
};
int main()
{
   programmer p;
   cout<<"Enter data"<<endl;
   p.getdata();
   cout<<endl<<"Displaying data"<<endl;
   p.display();
   getch();
   return 0;
}
```

Output

Enter data
Name: Karl Lens
Age: 31
Gender: Male
Name of Company: Dynamic Info
Salary: $21000
Number of programming language known: 4
Displaying data
Name: Karl Lens
Age: 31
Gender: Male
Name of Company: Dynamic Info
Salary: $21000
Number of programming language known: 4

This program is an example of multi level inheritance. Here, programmer class is derived from employee which is derived from person. Each class has required attributes and methods. The public features of person is inherited by employee and the public features of employee is inherited by programmer. The method getdata() asks user to input data, while display() displays the data.

4.1.4 Multiple Inheritance

Q6. Explain how to use multiple inheritance in C++ with an example.

Ans :

(Imp.)

When a class is derived from two or more base classes, such inheritance is called Multiple Inheritance. It allow us to combine the features of several existing classes into a single class.

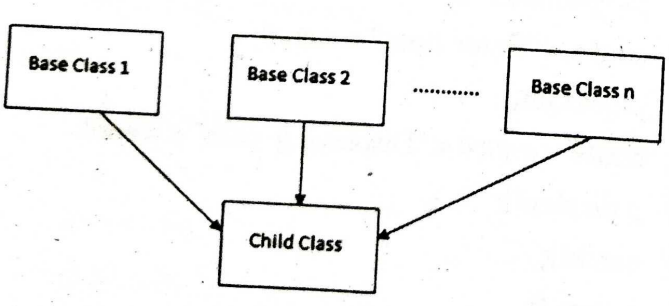


Fig.: Multiple Inheritance

For example,

- Petrol is derived from both liquid and fuel.
- A child has character of both his/her father and mother, etc

Syntax :

```
class base_class1
{
        properties;
        methods;
};
class base_class2
{
        properties;
        methods;
};
... ... ...
... ... ...
class base_classN
{
        properties;
        methods;
};
```

class derived_classname : visibility_mode base_class1, visibility_mode base_class2,... ,visibility_mode base_classN

```
{
    properties;
    methods;
};
```

Q7. What is ambiguity in multiple inheritance and how to resolve it.

Ans :

Ambiguity in Multiple Inheritance

In multiple inheritance, a single class is derived from two or more parent classes. So, there may be a possibility that two or more parents have same named member function. If the object of child class

needs to access one of the same named member function then it results in ambiguity. The compiler is confused as method of which class to call on executing the call statement.

// Demonstration of multiple inheritance

```
#include <iostream>
#include <conio.h>
using namespace std;
class A
{
    public:
        void display()
        {
            cout <<"This is method of A";
        }
};
class B
{
public:
    void display()
    {
        cout <<"This is method of B";
    }
};
class C: public A, public B
{
    public:
};
int main()
{
    C sample;
    sample.display();  /*causes ambiguity*/
     getch();
    return 0;
}
```

Ambiguity Resolution of Multiple Inheritance

This problem can be resolved by class name and using scope resolution operator to specify the class whose method is called.

Syntax :

derived_objectname.parent_classname:: same_named_function([parameter]);

In the above example, if we want to call the method of class A then we can call it as below, sample.A::display(); Similarly, if we need to call the method of class B then, sample.B::display();

4.1.5 Hierarchical Inheritance

Q8. What is hierarchical inheritance? Explain with an example.

Ans : **(Imp.)**

When more than one classes are derived from a single base class, such inheritance is known as Hierarchical Inheritance, where features that are common in lower level are included in parent class. Problems where hierarchy has to be maintained can be solved easily using this inheritance.

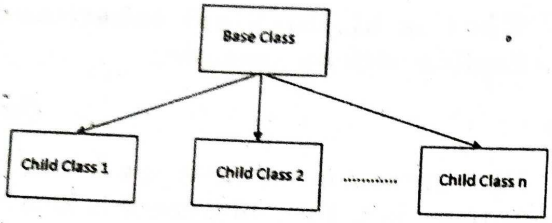
For example,

- Civil, Computer, Mechanical, Electrical are derived from Engineer.
- Natural language, Programming language are derived from Language.

Syntax

```
class base_classname
{
    properties;
    methods;
};
class derived_class1:visibility_mode base_classname
{
    properties;
    methods;
};
class derived_class2:visibility_mode base_classname
{
    properties;
    methods;
};
```

```
... ... ...
... ... ...
class derived_classN:visibility_mode base_classname
{
    properties;
    methods;
};
```



//C++ program to create Employee and Student inheriting from Person using Hierarchical Inheritance

```
#include <iostream>
#include <conio.h>
using namespace std;x
class person
{
  char name[100],gender[10];
  int age;
  public:
 void getdata()
 {
     cout<<"Name: ";
     fflush(stdin);  /*clears input stream*/
     gets(name);
     cout<<"Age: ";
     cin>>age;
     cout<<"Gender: ";
     cin>>gender;
 }
 void display()
 {
    cout<<"Name: "<<name<<endl;
    cout<<"Age: "<<age<<endl;
    cout<<"Gender: "<<gender<<endl;
 }
};
class student: public person
{
   char institute[100], level[20];
   public:
     void getdata()
     {
        person::getdata();
        cout<<"Name of College/School: ";
        fflush(stdin);
        gets(institute);
        cout<<"Level: ";
        cin>>level;
     }
     void display()
     {
        person::display();
          cout<<"Name of College/School:
          "<< institute<<endl;
        cout<<"Level: "<<level<<endl;
     }
};
class employee: public person
{
   char company[100];
   float salary;
   public:
     void getdata()
     {
        person::getdata();
        cout<<"Name of Company: ";
        fflush(stdin);
```

```
        gets(company);
        cout<<"Salary: Rs.";
        cin>>salary;
    }
    void display()
    {
        person::display();
            cout<<"Name of Company:
                "<<company<<endl;
            cout<<"Salary: Rs."<<salary
                <<endl;
    }
};
int main()
{
    student s;
    employee e;
    cout<<"Student"<<endl;
    cout<<"Enter data"<<endl;
    s.getdata();
    cout<<endl<<"Displaying data"<<endl;
    s.display();
    cout<<endl<<"Employee"<<endl;
    cout<<"Enter data"<<endl;
    e.getdata();
    cout<<endl<<"Displaying data"<<endl;
    e.display();
    getch();
    return 0;
}
```

Output

```
Student
Enter data
Name: John Wright
Age: 21
Gender: Male
Name of College/School: Abc Academy
Level: Bachelor
Employee
Enter data
Name: Mary White
Age: 24
Gender: Female
Name of Company: Xyz Consultant
Salary: $29000
Displaying data
Name: John Wright
Age: 21
Gender: Male
Name of College/School: Abc Academy
Level: Bachelor
Employee
Enter data
Name: Mary White
Age: 24
Gender: Female
Name of Company: Xyz Consultant
Salary: $29000
Displaying data
Name: Mary White
Age: 24
Gender: Female
Name of Company: Xyz Consultant
Salary: $29000
```

In this program, *student* and *employee* classes are derived from *person*. Person has two public methods: *getdata()* and *display()*. These methods are inherited by both *student* and *employee*. Input is given using *getdata()* method and displayed using *display()* method. This is an example of hierarchical inheritance since two classes are derived from a single class.

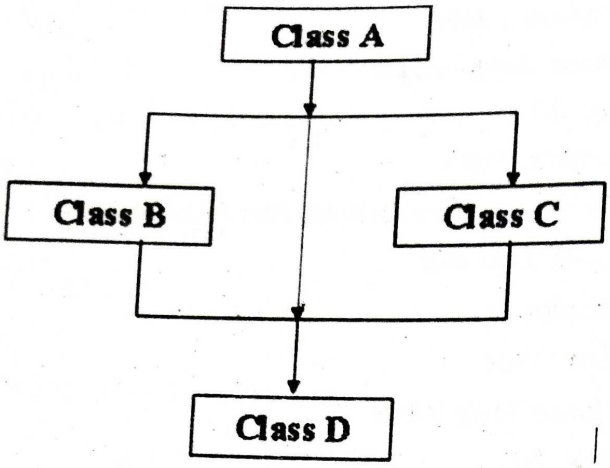
4.1.6 Hybrid Inheritance

Q9. What is Hybrid inheritance? Explain with an example. **(Imp.)**

Ans :

Combination of hierarchical and multiple inheritance is called hybrid inheritance.

In this lass D is derived from class B and Class C, Using class D object we can access all the members (A,B,C). But when we accessing Class A members we have two ways in this case we have an ambiguity, i.e. in which way we are getting the properties of class A, so here we implementing virtual keyword to avoid the ambiguity.




```
// demonstration of hybrid inheritance
#include
#include
class student
{
    protected:
    int rno;
    public:
    void getnum()
 {
 cout<<"enter the rno:";
 cin>>rno;
 }
 void putnum()
 {
    cout<<"\n"<
 }
};
class test: public student
{
  protected:
  int m1,m2;
  public:
  void getmarks()
  {
  getnum();
  cout<<"enter the mark1:";
  cin>>m1;
  cout<<"enter the mark2:";
  cin>>m2;
}
  void putmarks()
  {
     putnum();
     cout<<"\t"<
     cout<<"\t"<
  }
};
class sports
{
   protected:
   float score;
   public:
   void getscore()
   {
   cout<<"enter the score value:";
   cin>>score;
   }
   void putscore()
   {
   cout<<"\t"<
   }
};
class result: public test,public sports
{
   protected:
   float total;
   public:
   void display()
   {
   total=m1+m2+score;
   putmarks();
   putscore();
   cout<<"\t"<
  }
};
int main()
{
   clrscr();
   int i,n;
   result r[10];
```

```
cout<<"Enter the no of students:";
cin>>n;
for(i=0;i
{
  r[i].getmarks();
  r[i].getscore();
}
  cout<<"\nROLLNO"<<"\t MARK1"<<"\t
  MARK2"<<"\t SCORE"<<"\t TOTAL";
  for(i=0;i
{
  r[i].display();
}
  getch();
return 0;
}
```

OUPUT

```
Enter the no of students:3
Enter the rno:1
Enter the mark1:56
Enter the mark2:89
Enter the score value:58
Enter the rno:2
Enter the mark1:89
Enter the mark2:65
Enter the score value:75
Enter the rno:3
Enter the mark1:89
Enter the mark2:56
Enter the score value:74
STUDENT DETAILS
******************
ROLLNO MARK1 MARK2 SCORE TOTAL
1 56 89 58 203
2 89 65 75 229
3 89 56 74 219
```

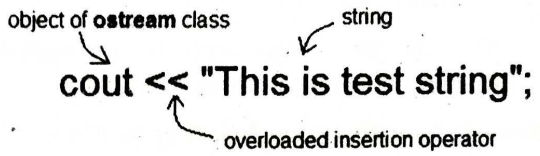
4.2 Operator Overloading

4.2.1 Rules for Overloading Operators

Q10. What is operator overloading? Write a syntax to overload operators.

Ans : **(Imp.)**

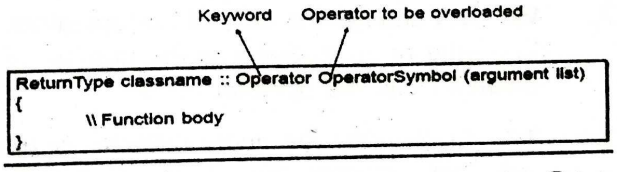
Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String (concatenation) etc.



Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded.Operator that are not overloaded are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Syntax



11. How to overload operators in C++ programming? Explain.

Ans :

To overload an operator, a special operator function is defined inside the class as:

```
class className
{
.....
```

```
public
    returnType operator symbol (arguments)
{
.....
}
.....
};
```

- Here, returnType is the return type of the function.
- The returnType of the function is followed by operator keyword.
- Symbol is the operator symbol you want to overload. Like: +, <, -, ++
- You can pass arguments to the operator function in similar way as functions.

Rules of Operator Overloading in C++

Operator overloading is a type of static or compile-time polymorphism. C++ supports the compile-time polymorphism. The function overloading and the operator overloading are common examples of compile-time polymorphism.

Let's see the rules for the operator overloading.

1. Only built-in operators like (+, -, *, /, etc) can be overloaded.
2. We cannot overload all of those operators that are not a part of C++ language like '$'.
3. We can't change the arity of the operators. The arity of an operator is the number of operands that the operator takes.
4. We can overload the unary operator as an only unary operator, and we cannot overload it as a binary operator and similarly, We can overload binary operators as an only binary operator, and we cannot overload it as a unary operator.
5. During the operator overloading, we cannot change the actual meaning of an operator. For example, We cannot overload the plus(+) operator to subtract one value form the other value.
6. The precedence of the operators remains the same during operator overloading.
7. The operator overloading is not possible for built-in data types. At least one user-defined data types must be there.
8. Some operators like assignment "=", address "&" and comma "," are by default overloaded.
9. When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.

List of operators that can be overloaded

List of operators that can be overloaded are mentioned below;

```
+ - * / % ^
& | ~ !, =
= ++ —
== != && ||
+= -= /= %= ^= &=
|= *= = [] ()
-> ->* new [] delete []
```

4.2.2 Overloading Unary Operators

Q12. What is unary operator overloading? Write a program to implement unary operator overloading.

Ans :

The unary operators operate on a single operand and following are the examples of Unary operators:

- The increment (++) and decrement (—) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj—.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
// demonstration of unary operator overloading
#include <iostream>
using namespace std;
class Distance

{
   private:
      int feet;     // 0 to infinite
      int inches;  // 0 to 12
   public:              // required constructors
   Distance(){
      feet = 0;
      inches = 0;
   }
   Distance(int f, int i){
      feet = f;
      inches = i;
   }
   // method to display distance
   void displayDistance()
   {
   cout << "F: " << feet << " I:" << inches
   <<endl;
   }
   // overloaded minus (-) operator
   Distance operator- ()
   {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
   }
   };
   int main()
   {
   Distance D1(11, 10), D2(-5, 11);
   -D1;                    // apply negation
   D1.displayDistance(); // display D1
   -D2;                    // apply negation
   D2.displayDistance(); // display D2
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

OUTPUT

```
F: -11 I:-10
F: 5 I:-11
```

4.2.3 Overloading Binary Operators

Q13. What is binary operator over loading? Write a c++ program to overload binary operator.

Ans : **(Imp.)**

You overload a binary operator with either a nonstatic member function that has one parameter, or a nonmember function that has two parameters. Suppose a binary operator @ is called with the statement t @ u, where t is an object of type T, and u is an object of type U. A nonstatic member function that overloads this operator would have the following form:

return_type operator@(U)

A nonmember function that overloads the same operator would have the following form:

return_type operator@(T, U)

```
#include<iostream>
#include<conio.h>
//Standard namespace declaration
using namespace std;
class overloading
{
   int value;
   public:
   void setValue(int temp)
   {
      value = temp;
   }
   overloading operator+(overloading ob)
   {
   overloading t;
   t.value=value+ob.value;
   return(t);
   }
```

```
    void display()
    {
        cout<<value<<endl;
    }
};
    //Main Functions
int main()
{
    overloading obj1,obj2,result;
    int a,b;
        cout<<"Enter the value of Complex
              Numbers a,b:";
        cin>>a>>b;
        obj1.setValue(a);
        obj2.setValue(b);
        result = obj1+obj2;
        cout<<"Input Values:\n";
        obj1.display();
        obj2.display();
        cout<<"Result:";
        result.display();
        getch();
        return 0;
}
```

Sample Output

```
Enter the value of Complex Numbers a,b:10
5
Input Values:
10
5
Result:15
```

4.3 POINTERS

4.3.1 Introduction to Pointers

Q14. What is pointer? Write about it.

Ans : **(Imp.)**

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it.

Pointer Value

Whenever a variable is declared, system will allocate a location to that variable in the memory, to hold value. This location will have its own address number.

For example, we declare a variable of type integer with the name a by writing:

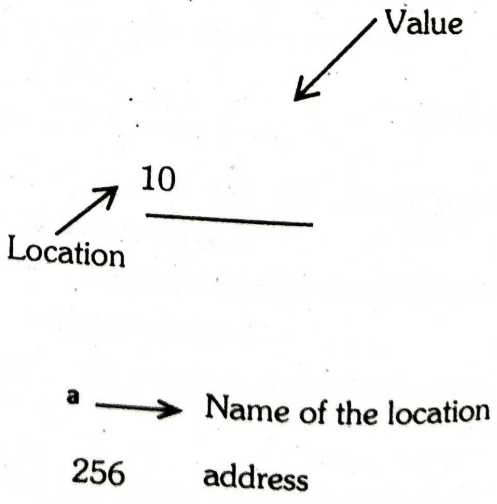
```
int a = 10;
```

On seeing the "int" part of this statement the compiler reserves 2 bytes of memory from the addresses to hold the value of the integer.

The value 10 will be placed in that memory location reserved for a.

These memory locations assigned to the variables by the system are called pointer values. The address 256is assigned to the variable a is a pointer value.

Let us assume that system has allocated memory location 256 for a variable **a**, which is called a pointer value.



The Address (&) Operator

The address of the variable cannot be accessed directly.

1. Pointer address operator is denoted by '&' symbol
2. When we use ampersand symbol as a prefix to a variable name '&', it gives the address of that variable.

The format specifier of address is %u(unsigned integer), because the addresses are always positive values.

Example: &a - It gives an address on variable a

```
// Demonstrating the address operator
#include<iostream.h>
void main()
{
    int n = 10;
    cout << "\nValue of a is : %d",a;
    cout << "\nValue of &a is : %u",&a;
}
```

Output

```
Value of a is : 10
Value of &a is : 256
```

Explanation

Consider the above example, where we have used to print the address of the variable using ampersand operator.

In order to print the variable we simply use name of variable while to print the address of the variable we use ampersand along with %u

Pointer Variable

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Value of pointer variable will be stored in another memory location.

The * Operator

1. In order to create pointer to a variable we use "*" operator .
2. '*' is called as 'Value at address' Operator
3. 'Value at address' Operator gives 'Value stored at Particular address.
4. 'Value at address' is also called as 'Indirection Operator' or 'Dereferencing Operator'

Consider the previous example, We can access the value 10 by either using the variable name a or the address 256. The variable that holds memory address are called pointer variables.

4.3.2 Declaring and Initializing Pointers

Q15. Explain how to access pointers with examples.

Ans : **(Imp.)**

Accessing a variable through Pointer

For accessing the xvariables through pointers, The following steps to be performed to access pointers.

- Declare a pointer variable
- Initializing of a variable to a pointer and
- Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Declaring a pointer variable

In C , every variable must be declared before they are used. A pointer variable can store only the address of the variable which has the same data-type as the pointer variable.

Syntax: data_type_name * variable name

Example: int *ptr;

Explanation: In the above example we have declare the pointer variable with the name of 'ptr' and its data-type in int, that means it can store the address of the variable which is of integer type.

This tells the compiler three things about the variable ptr.

1. The asterisk(*) tells that the variable ptr is a pointer variable.
2. ptr needs a memory location.
3. ptr points to a variable of type data type.

Different ways of declaring Pointer Variable:

Ex: Int *p;

Int *p;

Int *p;

****Note:** * can appears anywhere between Pointer_name and Data Type

Example of declaring integer pointer

```
int a= 10;
int *ptr
```

Example of declaring character pointer

```
char ch='A';
char *cptr;
```

Example of declaring float pointer

```
float pi= 3.14;
float *fptr;
```

Initializing Pointers

Once a pointer variable has been declared ,it must be assigned some value. The process of assigning the address of a variable to a pointer variable is known as initialization.

The initialization of the pointer variable is simple like other variable but in the pointer variable we assign the address instead of value.

Initialization of pointer can be done using 4 steps:

1. Declare a Pointer Variable and Note down the Data Type.
2. Declare another Variable with Same Data Type as that of Pointer Variable.
3. Initialize Ordinary Variable and assign some value to it.
4. Now Initialize pointer by assigning the address of ordinary variable to pointer variable.

Example :

```
int  *ptr,
int a = 10 ;
ptr = &a;
```

****Note**

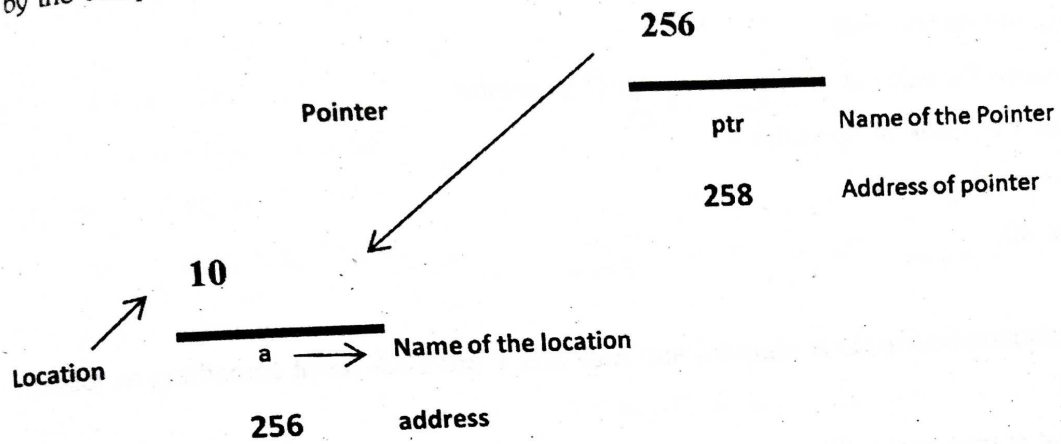
Pointers are always initialized before using it in the program

Explanation

Here in the above example we have a pointer variable *ptr* and another is a simple integer variable a, and we have assign the pointer variable with the address of the 'a'. That means the pointer variable 'ptr' is now has the address of the variable 'a'.

** **Note:** using the '*' asterisk sign before the pointer variable means the pointer variable is now pointing to the value at the location instead of the pointing to location.

Let us assume that system has allocated memory location 256 for a variable **a**, which is called a pointer value. Here the variable ptr is declared to hold the value of address of a variable 'a' which is allocated by the compiler.




```
// Demonstrating Declaring and Initialization of pointer
#include<iostream.h>
int main()
{
int a = 10;
int *ptr;
ptr = &a;
cout<< "\nValue of ptr : %u",ptr;
return(0);    }
```

Using Pointers in C++:

There are few important operations, which we will do with the pointers very frequently. (a) we define a pointer variables (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
// demonstration of how to use pointers
#include<iostream>
usingnamespace std;
int main ()
{
intvar=20;// actual variable declaration.
int*ip;// pointer variable
ip =&var;// store address of var in pointer variable
```

```
cout <<"Value of var variable: ";
cout <<var<< endl;
// print the address stored in ip pointer variable
cout <<"Address stored in ip variable: ";
cout << ip << endl;
// access the value at the address available in pointer
cout <<"Value of *ip variable: ";
cout <<*ip << endl;
return0;
}
```

When the above code is compiled and executed, it produces result something as follows:

OUTPUT

```
Value of var variable:20
Address stored in ip variable:0xbfc601ac
Value of *ip variable:20
```

4.3.3 Arithmetic Operations on Pointers

Q16. What is pointer arithmetic? Explain various types of pointer arithmetics with an example.

Ans : **(Imp.)**

Pointer arithmetic

We can perform arithmetic operations on pointer variable just as you can do with numeric value. As we know that, a pointer in C is a variable which is used to store the memory address of a numeric value. The arithmetic operations on pointer variable effects the memory address pointed by pointer.

Valid Pointer Arithmetic Operations

- Adding a number to pointer.
- Subtracting a number form a pointer.
- Incrementing a pointer.
- Decrementing a pointer.
- Subtracting two pointers.
- Comparison on two pointers.

Invalid Pointer Arithmetic Operations

- Addition of two pointers.
- Division of two pointers.
- Multiplication of two pointers.

Incrementing a Pointer

Incrementing Pointer is generally used in array because we have contiguous memory in array and we know the contents of next memory location.

Three rules should be used to increment a pointer

Address + 1 = Address

Address++ = Address

++Address = Address

Let ptr be an integer pointer which points to the memory location 5000 and size of an integer variable is 2 bytes. Now, when we increment pointer ptr,

```
ptr++;
```

It will point to memory location 5002 because it will jump to the next integer location which is 2 bytes next to the current location.

ptr

Incrementing a pointer is not same as incrementing an integer value. Incrementing Pointer Variable Depends Upon data type of the Pointer variable

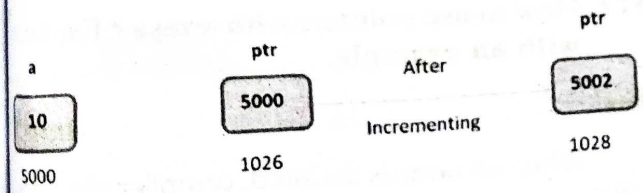


Fig.: Pictorial representation of incrementing a pointer

```
//Demonstration of incrementing a pointer
#include<stdio.h>
int main(){
int *ptr=(int *)5000;
double *ptr1=(double *)5020;
ptr=ptr+1;
cout << "New Value of Integer ptr : %u",ptr;
cout << "New Value of double ptr : %u",ptr1;
return 0;        }
```

OUTPUT

New Value of Integer Pointer : 5002

New Value of double Pointer : 5024

Decrementing Pointer

Similarly, Decrementing a pointer will decrease its value by the number of bytes of its data type.

Syntax: ptr—;

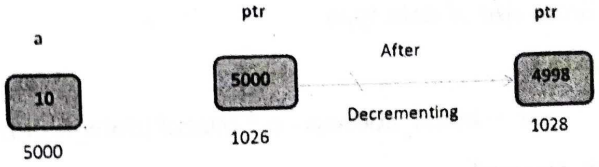


Fig.: Pictorial representation of decrementing pointer

Adding Numbers to Pointers

In C , we can add any integer number to Pointer variable. Adding a number N to a pointer leads the pointer to a new location after skipping N times size of data type.

Syntax:

final value = (address) + (number * size of data type)

For example, Let ptr be a 2-byte integer pointer, initially pointing to location 5000.

Then ptr + 5 = 5000 + 2*5 = 5010. Pointer ptr will now point at memory address 5010.

```
//demonstration of adding numbers to pointers
#include<stdio.h>
int main(){
int *ptr=(int *)5000;
ptr=ptr+5;
cout << "New Value of ptr : %u",ptr;
return 0;
}
```

OUTPUT

New Value of ptr :5010

Explanation

In the above program – int *ptr= (int *) 5000; will store 5000 in the pointer variable.

```
ptr= ptr   + 5 * (sizeof(integer))
   = 5000  + 5 * (2)
   = 5000  + 10
   = 5010
```

Subtracting Numbers from Pointers

Subtracting a number N from a pointers is similar to adding a number except in Subtraction the new location will be before current location by N times size of data type.

Syntax

ptr = initial_address - n * (sizeof (data_type))

For example

Let ptr be a 2-byte integer pointer, initially pointing to location 5000.

Then ptr - 3 = 5000 - 2*3 = 4994. Pointer ptr will now point at memory address 4994.

// Demonstration of Subtracting Numbers from a pointers

```
#include<iostream.h>
int main(){
int *ptr=(int *)5000;
ptr=ptr-3;
cout <<"New Value of ptr : %u",ptr;
return 0;
}
```

OUTPUT

New Value of ptr : 4994

Explanation : In the above example the following statement considers

```
ptr = ptr   - 3 * (sizeof(integer))
    = 5000  - 3 * (2)
    = 5000  - 6
    = 4994
```

Difference between two Pointers

We can take difference between two pointer which returns the number of bytes between the address pointed by both the pointers.

For example, if a pointer 'ptr1' points at memory location 10000 and pointer 'ptr' points at memory location 10008, the result of ptr2 - ptr1 is 8.

// Demonstration of difference between two pointers

```
#include<iostream.h>
int main()
{
int num , *ptr1 ,*ptr2 ;
ptr1 = &num ;
ptr2 = ptr1 + 2 ;
cout << "The difference between two pointers is : %d",ptr2 - ptr1;
return(0);
}
```

OUTPUT

The difference between two pointers is : 2

Explanation

In the above program the address of the variable num is stored in ptr1. Ptr1 is incremented by 2 and stored in ptr2.

4.3.4 Pointers with Arrays

Q17. How to use pointers with arrays ? Explain with an example.

Ans : **(Imp.)**

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. The base address is the location of the first element of the array.?The compiler defines the array name as a constant pointer to the first element.

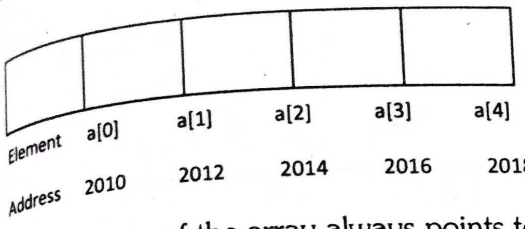
Pointers and One Dimensional Array

Suppose we declare an array a,

```
int a [5] = {1, 2, 3, 4, 5};
```

After declaring the array, the compiler creates an array with name a, the elements are stored in contiguous memory locations.

Assuming that the base address of a is 2010 and each integer requires two byte, the five element will be stored as follows.



In C, name of the array always points to the first element of an array. Here, address of first element of an array is & a[0].

Here variable a will give the base address, which is a constant pointer pointing to the element, a[0]. Therefore a is containing the address of a[0] i.e., 2050.

- Hence, & a[0] is equivalent to a.
- Also, value inside the address & a[0] and address a are equal.
- Value in address & a[0] is a[0] and value in address a is *a.
- Hence, a[0] is equivalent to *a.

Similarly

&a[1] is equivalent to (a+1) AND, a[1] is equivalent to *(a+1).

&a[2] is equivalent to (a+2) AND, a[2] is equivalent to *(a+2).

In C, you can declare an array and can use pointer to alter the data of an array.

Here in the above example We can declare a pointer of type int to point to the array **a.**

```
int *p;
p = a;
or p = &a[0]; //both the statements are
              // equivalent.
```

Now we can access every element of array a using p++ to move from one element to another.

```
//demonstrate pointer to arrays.
#include <iostream.h>
int main()
{
int i;
int a[5] = {1, 2, 3, 4, 5};
int *p = a; // same as int*p = &a[0]
for (i=0; i<5; i++)
    {
        Cout << "%d", *p;
    p++;
    }
```

OUTPUT

1 2 3 4 5

The above program, the pointer *p will print all the values stored in the array one by one.

Pointers and Two dimensional array

A two dimensional array is an array of one dimensional arrays. A two-dimensional array is stored in the memory in a series i.e. the elements of first row and first column and then the elements of the second row and coloumn and so on.

Let us consider a two dimensional array is of form, a[i] [j]

Then following notations are used to refer the two-dimensional array elements,

a ⟶ points to the first row

a+i ⟶ points to ith row

*(a+i) ⟶ points to first element in the ith row

*(a+i) +j ⟶ points to jth element in the ith row

((a+i)+j) ⟶ value stored in the ith row and jth column

Here is the generalized form for using pointer with multidimensional arrays.

((ptr + i) + j)is same asa[i][j]

suppose we declare an array as follows.

Example

```
int a[3][4]= {{1,2,3,,4}, {5,6, 7,8},
              {9,10,11,12}};
```

Base Address

The elements of a will be stored as shown below.

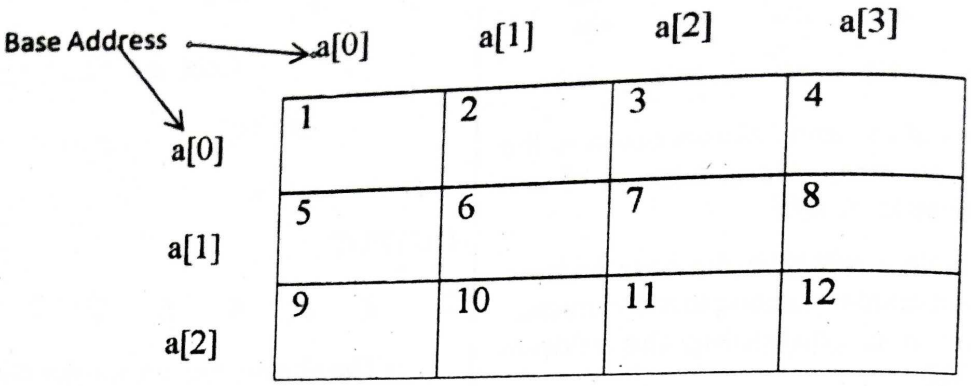


Fig.: Base Address = & a[0][0]

If we declare p as an int pointer with the initial address &a[0][0] then a[i][j] is equivalent to *(p+4xi+j)

If we increment i by 1, the p is incremented by 4, the size of each row making p element

a[2][3] is given by *(p+2x4+3) =*(p+11).

```
// demonstration of 2 d array using pointer
/* accessing elements of 2 d array using pointer*/
#include<iostream.h>
void main()
{
int z[3][3]={{1,2,3},{4,5,6},{7,8,9}};
int i,j;
int *p;
clrscr();
p=&z[0][0];
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
Cout << "\n %d\ti=%d j=%d\t%d",*(p+i*3+j),i,j,*(*(z+i)+j);
}
}
}
```

OUTPUT

1. i=0 j=0 1
2. i=0 j=1 2
3. i=0 j=2 3
4. i=1 j=0 4

5. i=1 j=1 5
6. i=1 j=2 6
7. i=2 j=0 7
8. i=2 j=1 8
9. i=2 j=2 9

Passing Array to a function

We know that the name of the array is a pointer to the first element, we can send the array name to the function for processing.

When we pass the array we do not use the address operator.

There are two possible ways to pass an array to a function, call by value and call by reference.

Declaration of function with array in parameter list

- int sum (int arr[]); //call by value method
- int sum (int* ptr); // call by reference method

Returning Array from function

We don't return an array from functions, rather we return a pointer holding the base address of the array to be returned.

```
int* sum (int x[])
    {
        //statements
        return x ;
    }
```

4.3.5 Arrays of Pointers

Q18. Write about array of pointers. **(Imp.)**

Ans :

Another useful structure that uses arrays and pointers is an array of pointers. This structure is especially helpful when number of elements in the array is variable.

Just like array of integers or characters, there can be array of pointers too.

An array of pointers can be declared as :

Syntax: <type> *<name>[<number-of-elements];

For example: char *ptr[3];

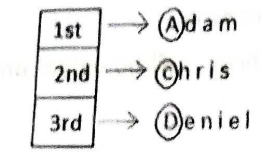
The above line declares an array of three character pointers.

Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3]={
        "Adam",
        "chris",
        "Deniel"
        };
```

```
//Now see same array without using pointer
char name[3][20]= {
        "Adam",
        "chris",
      "Deniel"
    };
```

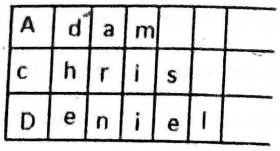
Using Pointer



char* name[3]

Only 3 locations for pointers, which will point to the first character of their respective strings.

Without Pointer



char name[3][20]

extends till 20 memory locations

In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.

4.3.6 Pointers to Objects

Q19. Demonstrate, how to use pointers with objects.

Ans :

(Imp.)

You can access an object either directly, or by using a pointer to the object.

To access an element of an object when using the actual object itself, use the dot operator.

To access a specific element of an object when using a pointer to the object, you must use the arrow operator.

To declare an object pointer, you use the same declaration syntax that you would use for any other pointer type.

Example

The next program creates a simple class called My_Class, defines an object of that class, called ob, and defines a pointer to an object of type My_Class, called p.

It then illustrates how to access ob directly, and how to use a pointer to access it indirectly.

```
// A simple example using an object pointer.
#include<iostream>
usingnamespace std;
class My_Class {//ww w. d e m o 2 s . c o m
int num;
public:
```

```
void set_num(int val) {num = val;}
void show_num();
};
void My_Class::show_num()
{
    cout << num <<"\n";
}
int main()
{
    My_Class ob, *p; // declare an object and pointer to it
    ob.set_num(1); // access ob directly
    ob.show_num();
    p = &ob; // assign p the address of ob
    p->show_num(); // access ob using pointer
    return 0;
}
```

Notice that the address of ob is obtained by using the & (address of) operator in the same way that the address is obtained for any type of variable.

When a pointer is incremented or decremented, it is increased or decreased in such a way that it will always point to the next element of its base type.

The same thing occurs when a pointer to an object is incremented or decremented: the next object is pointed to.

To illustrate this, the preceding program has been modified here so that ob is a two-element array of type My_Class.

Notice how p is incremented and decremented to access the two elements in the array.

```
// Incrementing and decrementing an object pointer.
#include<iostream>
usingnamespace std;
class My_Class {// w  ww . d e  m o  2s  . c o m
int num;
public:
void set_num(int val) {num = val;}
void show_num();
};
void My_Class::show_num()
{
  cout << num <<"\n";
}
int main()
{
```

```
    My_Class ob[2], *p;
    ob[0].set_num(10);
        // access objects directly
    ob[1].set_num(20);
    p = &ob[0];
        // obtain pointer to first element
    p->show_num();
        // show value of ob[0] using pointer
    p++; // advance to next object
    p->show_num();
        // show value of ob[1] using pointer
    p–; // retreat to previous object
    p->show_num();
        // again show value of ob[0]
    return 0;
}
```

4.3.7 This Pointer

Q20. What is this pointer? Explain with example.

Ans : **(Imp.)**

In C++ programming, this is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used to pass current object as a parameter to another method.
- It can be used to refer current class instance variable.
- It can be used to declare indexers.

Example

```
#include <iostream>
using namespace std;
class Employee {
    public:
    int id; //data member (also instance
            //variable)
    string name;
    // data member(also instance variable)
    float salary;
    Employee(int id, string name, float
            salary)
    {
        this->id = id;
        this->name = name;
        this->salary = salary;
    }
    void display()
    {
    cout<<id<<" "<<name<<"
        "<<salary <<endl;
    }
};
int main(void) {
Employee e1 =Employee(101, "Sonoo",
890000);
//creating an object of Employee
Employee e2=Employee(102, "Nakul",
59000);
//creating an object of Employee
e1.display();
e2.display();
return 0;
}
```

Output

```
101 Sonoo 890000
102 Nakul 59000
```

Short Question and Answers

1. Inheritance

Ans :

The process of obtaining the data members and methods from one class to another class is known as **inheritance**. It is one of the fundamental features of object-oriented programming.

Important points

- In the inheritance the class which is give data members and methods is known as base or super or parent class.
- The class which is taking the data members and methods is known as sub or derived or child class.

Syntax

```
class subclass_name : superclass_name
{
    // data members
    // methods
}
```

2. Types of Inheritance.

Ans :

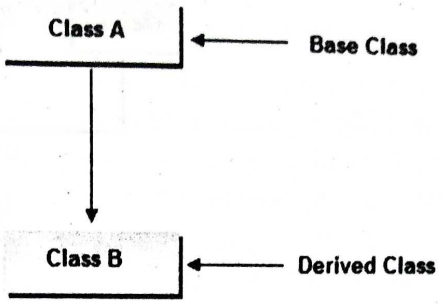
Types of Inheritance

Based on number of ways inheriting the feature of base class into derived class it have five types they are:

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

1. Single inheritance

In single inheritance there exists single base class and single derived class.



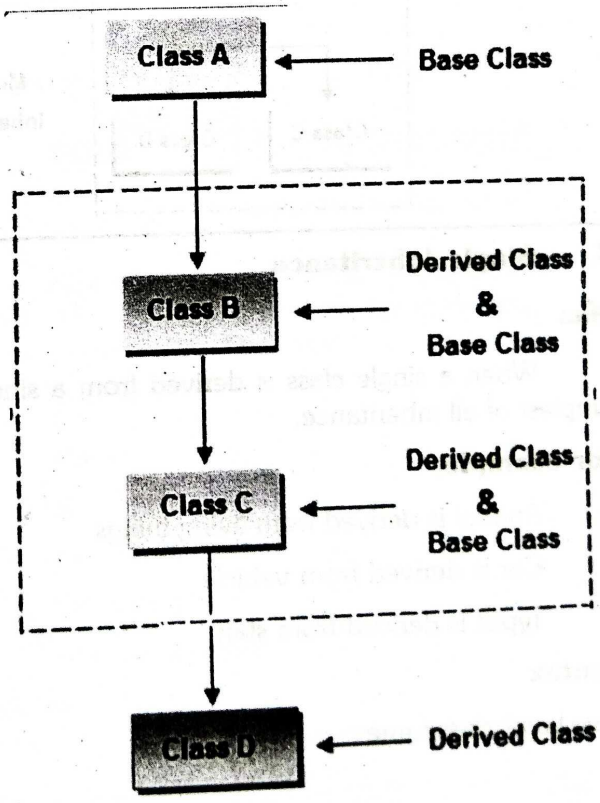
2. Multi level inheritances

In multi level inheritance there exists single base class, single derived class and multiple intermediate base classes.

Single base class + single derived class + multiple intermediate base classes.

Intermediate base classes

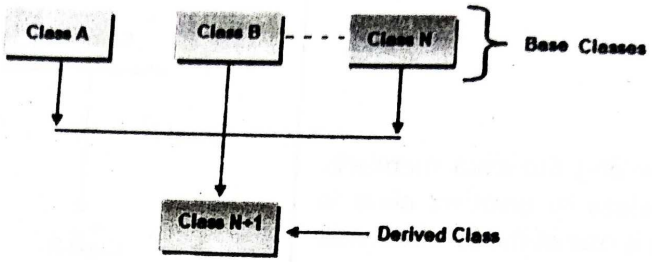
An intermediate base class is one in one context with access derived class and in another context same class access base class.



Hence all the above three inheritance types are supported by both classes and interfaces.

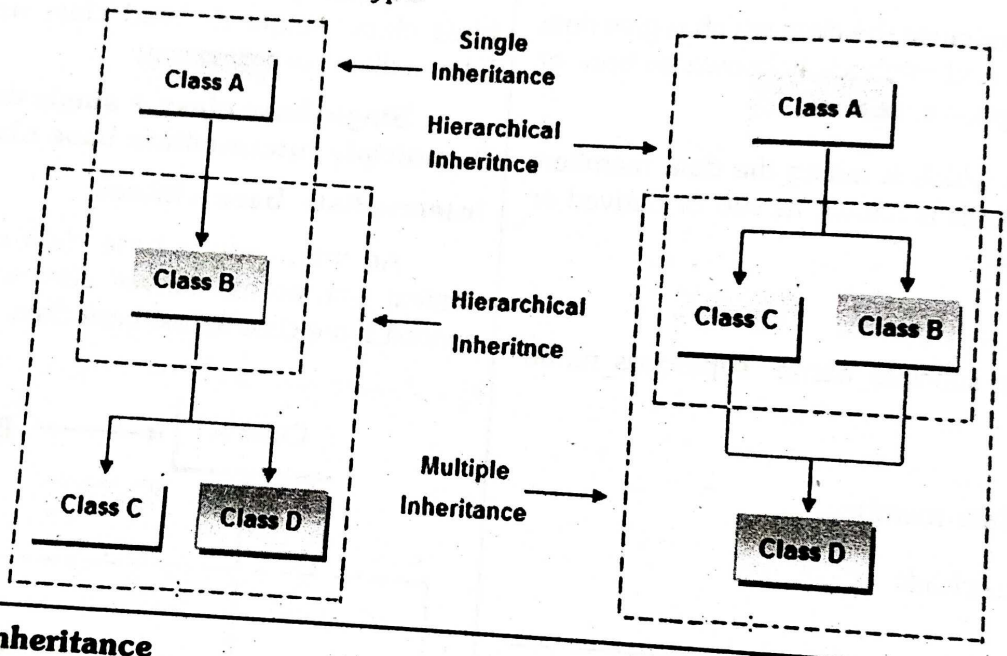
3. Multiple inheritance

In multiple inheritance there exist multiple classes and single derived class.



4. Hybrid inheritance

Combination of any inheritance type:



3. Single Inheritance

Ans :

When a single class is derived from a single parent class, it is called Single inheritance. It is the simplest of all inheritance.

For example,

- Animal is derived from living things
- Car is derived from vehicle
- Typist is derived from staff

Syntax

```
class base_classname
{
    properties;
    methods;
```

```
class derived_classname : visibility_mode
base_classname
{
   properties;
   methods;
};
```

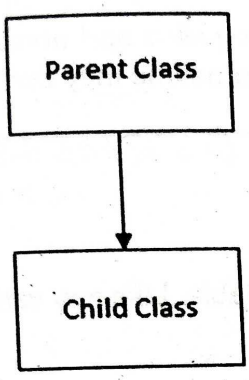


Fig.: Single Inheritance

4. Multiple Inheritance

Ans :

When a class is derived from two or more base classes, such inheritance is called Multiple Inheritance. It allow us to combine the features of several existing classes into a single class.

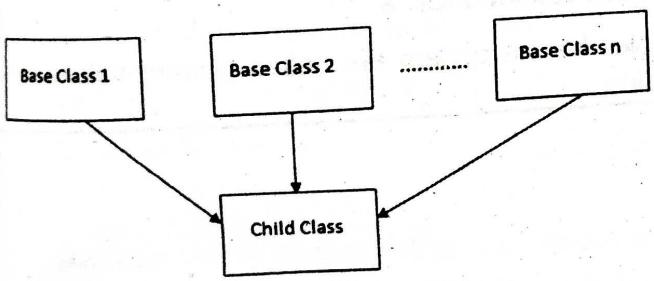


Fig.: Multiple Inheritance

5. Ambiguity in Multiple Inheritance

Ans :

In multiple inheritance, a single class is derived from two or more parent classes. So, there may be a possibility that two or more parents have same named member function. If the object of child class needs to access one of the same named member function then it results in ambiguity. The compiler is confused as method of which class to call on executing the call statement.

6. Hybrid inheritance.

Ans :

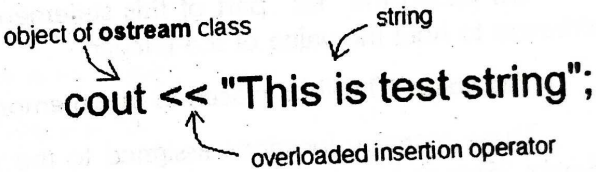
Combination of hierarchical and multiple inheritance is called hybrid inheritance.

In this lass D is derived from class B and Class C, Using class D object we can access all the members (A,B,C). But when we accessing Class A members we have two ways in this case we have an ambiguity, i.e. in which way we are getting the properties of class A, so here we implementing virtual keyword to avoid the ambiguity.

7. What is operator overloading? Write a syntax to overload operators.

Ans :

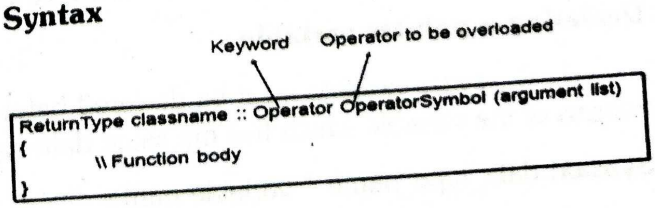
Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String (concatenation) etc.



Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded.Operator that are not overloaded are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Syntax



8. Overloading Unary operators.

Ans :

The unary operators operate on a single operand and following are the examples of Unary operators:

- The increment (++) and decrement (—) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj—.

9. What is pointer?

Ans :

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it.

Pointer Value

Whenever a variable is declared, system will allocate a location to that variable in the memory, to hold value. This location will have its own address number.

For example, we declare a variable of type integer with the name a by writing:

```
int a = 10;
```

On seeing the "int" part of this statement the compiler reserves 2 bytes of memory from the addresses to hold the value of the integer.

The value 10 will be placed in that memory location reserved for a.

These memory locations assigned to the variables by the system are called pointer values. The address 256is assigned to the variable a is a pointer value.

10. Declaring and Initializing Pointers

Ans :

Accessing a variable through Pointer

For accessing the variables through pointers, The following steps to be performed to access pointers.

- Declare a pointer variable
- Initializing of a variable to a pointer and
- Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Declaring a pointer variable

In C , every variable must be declared before they are used. A pointer variable can store only the address of the variable which has the same data-type as the pointer variable.

Syntax: data_type_name * variable name

11. How to use pointers with arrays ? Explain with an example.

Ans :

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. The base address is the location of the first element of the array.?The compiler defines the array name as a constant pointer to the first element.

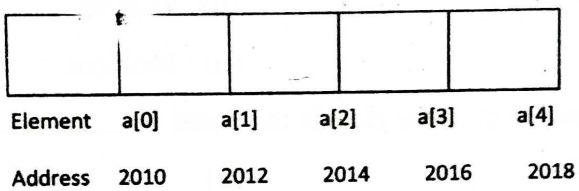
Pointers and One Dimensional Array

Suppose we declare an array a,

int a [5] = {1, 2, 3, 4, 5};

After declaring the array, the compiler creates an array with name a, the elements are stored in contiguous memory locations.

Assuming that the base address of a is 2010 and each integer requires two byte, the five element will be stored as follows.



In C, name of the array always points to the first element of an array. Here, address of first element of an array is & a[0].

Here variable a will give the base address, which is a constant pointer pointing to the element, a[0]. Therefore a is containing the address of a[0] i.e., 2050.

- Hence, & a[0] is equivalent to a.
- Also, value inside the address & a[0] and address a are equal.
- Value in address & a[0] is a[0] and value in address a is *a.
- Hence, a[0] is equivalent to *a.

Q12. Array of pointers.

Ans :

Another useful structure that uses arrays and pointers is an array of pointers. This structure is especially helpful when number of elements in the array is variable.

Just like array of integers or characters, there can be array of pointers too.

An array of pointers can be declared as :

Syntax: <type> *<name>[<number-of-elements];

Choose the Correct Answers

1. In Multipath inheritance, in order to remove duplicate set of records in child class, we _____ [c]

   (a) Write Virtual function in parent classes
   (b) Write virtual functions is base class
   (c) Make base class as virtual base class
   (d) All of these

2. When a child class inherits traits from more than one parent class, this type of inheritance is called _____ inheritance. [d]

   (a) Hierarchical (b) Hybrid
   (c) Multilevel (d) Multiple

3. The derivation of Child class from Base class is indicated by _____ symbol. [c]

   (a) :: (b) :
   (c) ; (d) |

4. Which is the correct example of a unary operator? [c]

   (a) & (b) ==
   (c) — (d) /

5. Which is called ternary operator? [a]

   (a) ?: (b) &&
   (c) ||| (d) ===

6. What is the syntax of overloading operator + for class A? [a]

   (a) A operator+(argument_list){}
   (b) A operator[+](argument_list){}
   (c) int +(argument_list){}
   (d) int [+](argument_list){}

7. Which of the following operator cannot be overloaded? [b]

   (a) + (b) ?:
   (c) – (d) %

8. Which of the following operator cannot be used to overload when that function is declared as friend function? [d]

   (a) -= (b) ||
   (c) == (d) []

What is the meaning of the following declaration? [b]

int(*p[5])();

(a) p is pointer to function

(b) p is array of pointer to function

(c) p is pointer to such function which return type is the array

(d) p is pointer to array of function

10. Referencing a value through a pointer is called [b]

(a) Direct calling

(b) Indirection

(c) Pointer referencing

(d) All of the above

Fill in the Blanks

1. Reusability of the code can be achieved in CPP through _____.
2. In case of inheritance where both base and derived class are having constructors, when an object of derived class is created then _____.
3. If the derived class is struct, then default visibility mode is _____.
4. class X, class Y and class Z are derived from class BASE. This is _____ inheritance.
5. The _____ and _____ of operators remains the same after and before operator overloading.
6. _____ operator can be used to overload when that function is declared as friend function?
7. _____ operators should be preferred to overload as a global function rather than a member method?
8. When overloading unary operators using Friend function,it requires _____ argument/s.
9. Generic pointers can be declared with _____ .
10. _____ operator returns the address of unallocated blocks in memory?

Answers

1. Inheritance
2. Constructor of base class will be executed first followed by derived class
3. Public
4. Hierarchical
5. Precedence and associativity
6. |=
7. Insertion Operator <<
8. One
9. Void
10. New