Overloading and Inheritance

Introduction & Need of Operator Overloading:

- C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.
- An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).
- When you call an overloaded function or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions.
- The process of selecting the most appropriate overloaded function or operator is called overload resolution.
- Operator overloading is one of the special feature of C++.
- It also shows the extensibility of C++. C++ permits us to add two variables of user defined types with the same way that is applied with built in type data type.
- This refers to ability to provide special meaning for existing data type.
- This mechanism of giving such special meaning to an operator is known as overloading.
- Operator overloading provides a flexible option for creation of new definition for most of the C++ operators.
- We can assign additional meaning to all existing C++ operators except following:
  1. Class member access operators (.,*)
  2. Scope resolution operator (::)
  3. Size of operator (sizeof)
  4. Conditional operator (? ;)
- These operators are attributed to fact that an operator takes names as their operand instead of values.
- Note: When an operator is overloaded, its original meaning is not lost. E.g. the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

Definition Operator Overloading:

- To define an additional task to an operator, we must specify what it means in relation to class to which the operator is applied.
- This is done with the help of a special function, called operator function which describes the task.

```c
return type class_name :: operator op( argument list)
{
    // Body of the function
    // The task defined by overloaded operator
}
```

where:
- return type is type of value returned by specified operation.
Operator function is no static member function or it may be friend function.
A basic difference between them is that friend function will have only one argument for unary and binary operator whereas member function has no argument for unary operators and only one for binary operators.
This is because the object used to invoke the member function is passé implicitly and therefore is available for member function.
This is not the case with friend functions. Arguments may be passed either by value or by reference.
For defining an additional task to an operator, we must mention what is means in relation to the class to which it (the operator) is applied.
The operator function helps us in doing so.
The Syntax of declaration of an Operator function is as follows:

operator operator_name

For example, suppose that we want to declare an Operator function for ‘=’, We can do it as follows:

operator =

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.
A Binary Operator can be defined either a member function taking one argument or a global function taking one arguments.
For a Binary Operator X, a X b can be interpreted as either an operator X (b) or operator X (a, b). For a Prefix unary operator Y, Ya can be interpreted as either a.operator Y ( ) or Operator Y (a).
For a Postfix unary operator Z, aZ can be interpreted as either a.operator Z(int) or Operator (Z(a),int).
The operator functions namely operator=, operator [ ], operator ( ) and operator? must be non-static member functions. Due to this, their first operands will be lvalues.
Following example explains how an assignment operator can be overloaded.

Program for illustrating Assignment Operator
#include <iostream>
using namespace std;

class Distance
{
private:
    int feet; // 0 to infinite
    int inches; // 0 to 12
public:
    // required constructors
    Distance()
        feet = 0;

    operator =
    { ... }
inches = 0;
}
Distance(int f, int i){
    feet = f;
    inches = i;
}
void operator=(const Distance &D )
{
    feet = D.feet;
    inches = D.inches;
}
// method to display distance
void displayDistance()
{
    cout << 'F: " << feet << " I:" << inches << endl;
}

};
int main()
{
    Distance D1(11, 10), D2(5, 11);
    cout << "First Distance : ";
    D1.displayDistance();
    cout << "Second Distance : ";
    D2.displayDistance();

    // use assignment operator
    D1 = D2;
    cout << "First Distance : ";
    D1.displayDistance();

    return 0;
}

When the above code is compiled and executed, it produces the following result:
First Distance : F: 11 I:10
Second Distance : F: 5 I:11
First Distance : F: 5 I:11

- An operator function should be either a member or take at least one class object argument. The operators new and delete need not follow the rule.
- Also, an operator function, which needs to accept a basic type as its first argument, cannot be a member function. Some examples of declarations of operator functions are given below:
class P
{
    P operator ++ (int);//Postfix increment
    P operator ++ ( ); //Prefix increment
    P operator || (P); //Binary OR
Some examples of Global Operator Functions are given below:

- P operator – (P); // Prefix Unary minus
- P operator – (P, P); // Binary “minus”
- P operator – - (P & int); // Postfix Decrement

We can declare these Global Operator Functions as being friends of any other class.

Examples of operator overloading:

- **Operator overloading using friend.**
  
  ```cpp
  Class time
  {
    int r;
    int i;
    public:
      friend time operator + (const time &x, const time &y);
      // operator overloading using friend
      time () { r = i = 0;}
      time (int x, int y) {r = x; i = y;}
  }
  
  time operator + (const time &x, const time &y)
  {
    time z;
    z.r = x.r + y.r;
    z.i = x.i + y.i;
    return z;
  }

  main()
  {
    time x, y, z;
    x = time (5, 6);
    y = time (7, 8);
    z = time (9, 10);
    z = x+y; // addition using friend function +
  }
  ```

Unary operators overloading in C++:

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.
Program for illustrating Minus(-) 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:
F: -11 I:-10
F: 5 I:-11

Programming for Overloading Increment ++ and Decrement --
#include <iostream>
using namespace std;

5 | P a g e
class Time
{
private:
    int hours;             // 0 to 23
    int minutes;           // 0 to 59
public:
    // required constructors
    Time()
    {
        hours = 0;
        minutes = 0;
    }
    Time(int h, int m)
    {
        hours = h;
        minutes = m;
    }
    // method to display time
    void displayTime()
    {
        cout << "H: " << hours << " M:" << minutes << endl;
    }
    // overloaded prefix ++ operator
    Time operator++ ()
    {
        ++minutes;          // increment this object
        if(minutes >= 60)
        {
            ++hours;
            minutes -= 60;
        }
        return Time(hours, minutes);
    }
    // overloaded postfix ++ operator
    Time operator++( int )
    {
        // save the original value
        Time T(hours, minutes);
        // increment this object
        ++minutes;
        if(minutes >= 60)
        {
            ++hours;
            minutes -= 60;
        }
        // return old original value
        return T;
    }
};
int main()
{

Time T1(11, 59), T2(10,40);

++T1;                   // increment T1
T1.displayTime();        // display T1
++T1;                    // increment T1 again
T1.displayTime();        // display T1

T2++;                    // increment T2
T2.displayTime();        // display T2
T2++;                    // increment T2 again
T2.displayTime();        // display T2
return 0;
}

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

H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42

BINARY OPERATOR OVERLOADING:-

In overloading binary operator, a friend function will have two arguments, while a member function will have one argument.

Following example shows how to overload + operator to add 2 complex number

Program for adding Two Complex Numbers
#include<iostream.h>
class complex
{
  float x, y;
  public :
  complex ();
  complex (float real, float imag)
  {
    x = real;
    y = imag;
  }
  complex operator +(complex);
  void display(void);
};
complex complex: operator+(complex c)
{
  complex temp;
  temp.x = x + c.z;
  temp.y = y + c.y;
  return (temp);
void complex::display(void)
{
    cout « z < i + j « y < i « i;
}
main()
{
    complex c1,c2,c3;
    c1 = complex (2.5, 3.5);
    c2 = complex (1.6, 2.7);
    c3 = c1 + c2;
    cout « c1 = i; c1.display();
    cout « c2 = i; c2.display();
    cout « c3 = i; c3.display();
}
output:
c1 = 2.5 + j 3.5
c2 = 1.6 + j 2.7
c3 = 4.1 + j 6.2

In the above program, the function operator +, is expected to add two complex values and return a complex value as a result but receives only one value as argument.

The function is executed by the statement
c3 = c1 + c2

Here, the object c1 is used to invoke the function and c2 plays the role of an argument that is passed to the function. The above statement is equivalent to

c3 = c1.operator +(c2)

Here, in the operator + () function, the data members of c1 are accessed directly and the data member of c2 are accessed using the dot operator. Thus in overloading binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument.

OVERLOADING BINARY OPERATORS USING FRIENDS

- Friend functions may be used in the place of member functions for overloading a binary operator.
- The only difference being that a friend function requires two arguments to be explicitly passed to it while a member function requires only one.
- The same complex number program with friend function can be developed as friend complex operator +(complex, complex); and we will define this function as

complex operator +(complex a, complex b)
{
    return complex (c.x + b.x), (a.y + b.y);
}
in this case, the statement
\[ c3 = c1 + c2; \]
is equal to \( c3 = \text{operator} + (c1, c2) \)

- In certain situation it is require using a friend function rather than member function.
- Example, If we need to use two different types of operands for a binary operator, i.e. one is an object and another is a built-in type data.

Example:- \( A = B + 2; \)
Where \( A \) & \( B \) are objects of same class. This will work for member function, but \( A = 2 + B; \) Will not work.
Because, here left-hand operand which is responsible for invoking the member function should be an object of the same class.

**RULES FOR OVERLOADING OPERATORS:**
1. Only existing operators can be overloaded. New operators cannot be overloaded.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. We cannot change the basic meaning of an operator. That is to say, We cannot redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded like size of operator(sizeof), membership operator(·), pointer to member operator(·*), scope resolution operator(::), conditional operators(?) etc.
6. We cannot use “friend” functions to overload certain operators. However, member function can be used to overload them. Friend Functions can not be used with assignment operator(=), function call operator(), subscripting operator([]), class member access operator(->) etc.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, -, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Operator overloading using member function:
```cpp
class abc {
    char * str;
    int len; // Present length of the string
    int max_length; // (maximum space allocated to string)
public:
    abc(); // black string of length 0 of maximum allowed length of size 10.
    abc(const abc &s); // copy constructor
    ~abc(); // delete str;
    int operator==(const abc &s) const; // check for equality
    abc & operator=(const abc &s);
};
```
// overloaded assignment operator
friend abc operator + (const abc &s1, const abc &s2);
} // string concatenation
abc:: abc()
{
    max_length = 10;
    str = new char [max_length];
    len = 0;
    str[0] = ‘\0’;
}
abc:: abc(const abc &s)
{
    len = s.len;
    max_length = s.max_length;
    str = new char [max_length];
    strcpy(str, s.str); // physical copying in the new location.
}

Note: Please note the need of explicit copy constructor as we are using pointers. For example, if a string object containing string “first” is to be used to initialise a new string and if we do not use copy constructor then will cause:

That is two pointers pointing to one instance of allocated memory, this will create problem if we just want to modify the current value of one of the string only. Even destruction of one string will create problem. That is why we need to create separate space for the pointed string as:

Thus, we have explicitly written the copy constructor. We have also written the explicit destructor for the class. This will not be a problem if we do not use pointers.

abc:: ~abc()
{
    delete str;
}
abc &abc::operator = (const abc &s)
{
    if (this != &s) // if the left and right hand variables are different
    {
        len = s.len;
        max_length = s.max_length;
        delete str; // get rid of old memory space allocated to this string
str = new char [max_length]; // create new locations
strcpy (str, s.str); // copy the content using string copy function
}
return *this;
}
// Please note the use of this operator which is a pointer to object that invokes the call to
// this assignment operator function.
inline int abc :: operator == (const abc &s ) const
{
  // uses string comparison function
  return strcmp (str,s.str);
} abc abc:: operator + (const abc &s)
abc s3;
s3.len = len + s.len;
s3.max_length = s3.len;
char * newstr = new char [length + 1];
strcpy (newstr, s.str);
strcat (newstr,str);
s3.str = newstr;
return (s3);

Overloading << operator: To overload << operator, the following function may be used:
Ostream & operator << (ostream &s, const abc &x )
{
  s<< “The String is:” <<x; }
return s;
}

You can write appropriate main function and use the above overloaded operators as shown in the
complex number example.

Type conversions:

What is type conversion?

It is the process of converting one type into another. In other words converting an expression of a
given type into another is called type casting.

How to achieve this?

There are two ways of achieving the type conversion namely:
• Automatic Conversion otherwise called as Implicit Conversion
• Type casting otherwise called as Explicit Conversion

Let us see each of these in detail:

• Automatic Conversion otherwise called as Implicit Conversion
• This is not done by any conversions or operators. In other words the value gets
  automatically converted to the specific type to which it is assigned.
Let us see this with an example:

```cpp
#include <iostream>
using namespace std;
void main()
{
    short x=6000;
    int y;
    y=x;
}
```

In the above example the data type short namely variable x is converted to int and is assigned to the integer variable y.

So as above it is possible to convert short to int, int to float and so on.

**Type casting otherwise called as Explicit Conversion**

- Explicit conversion can be done using type cast operator and the general syntax for doing this is:
  
  ```cpp
  datatype (expression);
  ```

- Here in the above datatype is the type which the programmer wants the expression to get changed as.

- In C++ the type casting can be done in either of the two ways mentioned below namely:
  
  - C-style casting
  - C++-style casting

**The C-style casting takes the syntax as**

- ```cpp
  (type) expression
  ```

  This can also be used in C++.

- Apart from the above, the other form of type casting that can be used specifically in C++ programming language namely C++-style casting is as below namely:
  
  ```cpp
  type (expression)
  ```

- This approach was adopted since it provided more clarity to the C++ programmers rather than the C-style casting.

- Say for instance the as per C-style casting
  
  ```cpp
  (type) firstVariable * secondVariable
  ```

  is not clear but when a programmer uses the C++ style casting it is much more clearer as below

  ```cpp
  type (firstVariable) * secondVariable
  ```

**TYPE CONVERSIONS**

We have overloaded several kinds of operators but we haven’t considered the assignment operator (=). It is a very special operator having complex properties. We know that = operator assigns values from one variable to another or assigns the value of user defined object to another of the same type. For example,

```cpp
int x, y;
x = 100;
y = x;
```

Here, first 100 is assigned to x and then x to y.

Consider another statement, 13 = t1 + t2;
This statement used in program earlier, assigns the result of addition, which is of type time to object t3 also of type time. So the assignments between basic types or user defined types are taken care by the compiler provided the data type on both sides of = are of same type.

But what to do in case the variables are of different types on both sides of the = operator? In this case we need to tell to the compiler for the solution.

Three types of situations might arise for data conversion between different types:

(i) Conversion from basic type to class type.
(ii) Conversion from class type to basic type.
(iii) Conversion from one class type to another class type.

Now let us discuss the above three cases:

(i) Basic Type to Class Type: This type of conversion is very easy. For example, the following code segment converts an int type to a class type.

```cpp
class distance
{
int feet;
int inches;
public:
    ...
    ...
distance (int dist) //constructor
    {
    feet = dist/12;
inches = dist%12;
    }
};
```

The following conversion statements can be coded in a function:

distance dist1; //object dist1 created
int length = 20;
dist1=length; //int to class type

After the execution of above statements, the feet member of dist1 will have a value of 1 and inches member a value of 8, meaning 1 feet and 8 inches.

• A class object has been used as the left hand operand of = operator, so the type conversion can also be done by using an overloaded = operator in C++.

(ii) Class Type to Basic Type: For conversion from a basic type to class type, the constructors can be used. But for conversion from a class type to basic type constructors do not help at all. In C++, we have to define an overloaded casting operator that helps in converting a class type to a basic type. The syntax of the conversion function is given below:

```cpp
Operator typename()
{
    ....
    .... //statements
}
```

Here, the function converts a class type data to typename. For example, the operator float () converts a class type to type float, the operator int () converts a class type object to type int. For example,

matrix :: operator float ()
Here, the function finds the norm of the matrix (Norm is the square root of the sum of the squares of the matrix elements). We can use the operator float () as given below:
float norm = float(arr);
or
float norm = arr;
where arr is an object of type matrix. When a class type to a basic type conversion is required, the compiler will call the casting operator function for performing this task. The following conditions should be satisfied by the casting operator function:
(a) It must not have any argument
(b) It must be a class member
(c) It must not specify a return type.

(iii) One Class Type to Another Class Type

There may be some situations when we want to convert one class type data to another class type data. For example,
Obj2 = obj1; //different type of objects
Suppose obj1 is an object of class studdata and obj2 is that of class result. We are converting the class studdata data type to class result type data and the value is assigned to obj2. Here studdata is known as source class and result is known as the destination class.
The above conversion can be performed in two ways:
(a) Using a constructor.
(b) Using a conversion function.

When we need to convert a class, a casting operator function can be used i.e. source class. The source class performs the conversion and result is given to the object of destination class. If we take a single-argument constructor function for converting the argument’s type to the class type (whose member it is). So the argument is of the source class and being passed to the destination class for the purpose of conversion. Therefore it is compulsory that the conversion constructor be kept in the destination class.
Inheritance:
Introduction

- Inheritance allows a class to include the members of other classes without repetition of members.
- There were three ways to inheritance means, “public parts of super class remain public and protected parts of super class remain protected.”
- Private Inheritance means “Public and Protected Parts of Super Class remain Private in Sub-Class”.
- Protected Inheritance means “Public and Protected Parts of Superclass remain protected in Subclass.
- One of the fundamental ideas behind object-oriented programming is that code should be reusable.
- However, existing code often does not do EXACTLY what you need it to.
- For example, what if you have a triangle and you need a square?
- In this case, we are presented with a number of choices on how to proceed, all of which have various benefits and downsides.
- Perhaps the most obvious way to proceed is to change the existing code to do what you want.
- However, if we do this, we will no longer be able to use it for its original purpose, so this is rarely a good idea.
- A slightly better idea is to make a copy of some or all of the existing code and change it to do what we want. However, this has several major demerits.
- First, although copy-and-paste seems simple enough, it’s actually quite dangerous.
- A single omitted or misplaced line can cause the program to work incorrectly and can take days to find in a complex program.
- Renaming a class via search-and-replace can also be dangerous if you inadvertently replace something you didn’t mean to.
- Second, to rewrite the code to make it do what you want, you need to have an intimate understanding what it does.
- This can be difficult when the code is complex and not adequately documented.
- Third, and perhaps most relevant, this generally involves duplicating of existing functionality, which causes a maintenance problem.
- Improvements or bug fixes have to be added to multiples copies of functions that do essentially the same thing, which wastes programmer time.
- And that’s assuming the programmer realizes multiple copies even exist! If not, some copies may not get the improvements or bug fixes.
- Inheritance solves most of these problems in an efficient way.
- Instead of manually copying and modifying every bit of code your program needs, inheritance allows you directly reuse existing code that meets your needs.
- You only need to add new features, redefine existing features that do not meet your needs, or hide features you do not want.
- This is typically much less work (as you are only defining what has changed compared to the base, rather than redefining everything), and safer too.
- Furthermore, any changes made to the base code automatically get propagated to the inherited code.
- This means it is possible to change one piece of code (e.g. to apply a bug fix) and all derived objects will automatically be updated.
CONCEPT OF INHERITANCE

- Inheritance is a concept which is the result of commonality between classes.
- Due to this mechanism, we need not repeat the declaration as well as member functions in a class if they are already present in another class.
- For example, consider the classes namely “minister” and “prime minister”.
- Whatever information is present in minister, the same will be present in prime minister also.
- Apart from that, there will be some extra information in class prime minister due to the extra privileges enjoyed by him.
- Now, due to the mechanism of inheritance, it is enough only to indicate that information which is specific to prime minister in its class.
- In addition, the class prime minister will inherit the information of class minister.

Base class and Derived Class:

- Let us take the classes, Employee and Manager.
- A Manager is an Employee with some additional information. When we are declaring the classes Employee and Manager without applying the concept of inheritance, they will look as follows:

  ```cpp
class Employee
  {
  public:
  char* name;
  int age;
  char* address;
  int salary;
  char*department;
  int id;
  };
  ```

- Now, the class Manager is as follows:

  ```cpp
class Manager
  {
  public:
  char* name;
  int age;
  char* address;
  int salary;
  char*department;
  int id;
  employee* team_members; //He heads a group of employees
  int level; //his position in hierarchy of the organisation
  ...}
  ```

- Now, without repeating the entire information of class Employee in class Manager, we can declare the Manager class as follows:

  ```cpp
class Manager: Public Employee
  {
  public:
  Employee* Team_members;
  int level;
  }
  ```
- The latest declaration of class Manager is the same as that of its previous one, with the exception that we did not repeat the information of class Employee explicitly.
- This is what is meant by the Application of inheritance mechanism. Please note that in the above example, Employee is called Base Class and Manager is called Derived Class.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.
- This existing class is called the base class, and the new class is referred to as the derived class.
- A derived class represents a more specialized group of objects.
- Typically, a derived class contains behaviours inherited from its base class plus additional behaviours.
- A class can be derived from more than one class, 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:

```
class derived-class : access-specifier base-class
```

Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

C++ offers three forms of inheritance—public, protected and private.

**Program illustrating the Concept of Inheritance**

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

// Base class
class Shape
{
    public:
        void setWidth(int w)
        {
            width = w;
        }
        void setHeight(int h)
        {
            height = h;
        }
    protected:
        int width;
        int height;
};

// Derived class
class Rectangle: public Shape
{
    public:
        int getArea()
        {
            // Base class
            // Derived class
        }
```
```cpp
int main(void)
{
    Rectangle Rect;
    Rect.setWidth(5);
    Rect.setHeight(7);
    cout << "Total area: " << Rect.getArea() << endl;  // Print the area of the object.
    return 0;
}
```

Access Control and Inheritance:
- A derived class can access all the non-private members of its base class.
- Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.
- We can summarize the different access types according to who can access them in the following way:

<table>
<thead>
<tr>
<th>Access</th>
<th>public</th>
<th>protected</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same class</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Derived classes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Outside classes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

- When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance.
- The type of inheritance is specified by the access-specifier as explained above. We hardly use protected or private inheritance, but public inheritance is commonly used.
- While using different type of inheritance, following rules are applied:
  - Public Inheritance: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.
  - Protected Inheritance: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.
  - Private Inheritance: When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Types of Inheritances:
- New classes can be built from the existing classes.
- It means that we can add additional features to an existing class without modifying it.
- The new class is referred as derived class or subclass and the original class is known as base classes or super class.
- Following are the types of Inheritance:
  I. Single Level Inheritance
II. Multiple Inheritances
III. Hierarchical inheritance
IV. Multilevel Inheritance
V. Hybrid Inheritance.

**Single Level Inheritance:**
- A derived class with only one base class is called single inheritance.
- Consider a simple example of single inheritance.
- In this program show a base class B and derived class D. The class B contains one private data member, one public data member, and three public member functions.
- The class D contains one private data members and two public member functions.

**Program For Illustrating Single Inheritance**

```cpp
#include <iostream.h>

Class B
{
    int a;
    public:
    int b;
    void get_ab();
    int get_a();
    void show_a();
};

Class D: public B
{
    int c;
    public:
    void mul();
    void display();
};

Void B :: get_ab()
{
    a=5;
    b=10;
}

int B :: get_a()
{
    return a;
}

void B :: show_a()
```
Multiple Inheritances:

- A class can inherit properties from more than one class which is known as multiple inheritances.
- This form of inheritance can have several super classes.
- A class can inherit the attributes of two or more classes as shown below diagram.
- Multiple inheritances allow us to combine the features of several existing classes as a starting point for defining new classes.
- It is like a child inheriting the physical features of one parent and the intelligent if another.

```cpp
#include <iostream.h>
class M {

};
```
protected:
int m;
public :
void get_m(int);
};
class N
{
protected:
int n;
public :
void get_n(int);
};
class P : public M, public N
{
public :
void display();
};
void M :: get_m(int x)
{
M=x;
}
void N::get_n(int y)
{
N=y;
}
void P :: display()
{
cout<<"m=",<<m<<"n";
cout<<"n=",<<n<<"n";
cout<<"m*n=",<<m*n<<"n";
}
int main()
{
P p1;
P1.get_m(10);
P1.get_n(20);
P1.display();
return 0;
}

Hierarchical Inheritances:
- When the properties of one class are inherited by more than one class, it is called hierarchical inheritance.
- This form has one super class and many Subclasses.
- More than one class inherits the traits of one class. For example: bank accounts.
Multilevel Inheritance:
- A class can be derived from another derived class which is known as multilevel inheritance.
- Order of Constructor Calling in Multilevel Inheritance, when the object of a subclass is created the constructor of the subclass is called which in turn calls constructor of its immediate super class.
- For example, if we take a case of multilevel inheritance, where class B inherits from class A, and class C inherits from class B, which show the order of constructor calling.

Hybrid Inheritance:
- There could be situations where we need to apply two or more types of inheritance to design one inheritance called hybrid inheritance.
- For instance, consider the case of processing the student results, the weight age for sport is stored in separate classes.
- Example of Hybrid Inheritance:

```cpp
class stud
{
    Protected:
    int rno;
    Public:
    Void getno(int n)
{
```
Rno=n;
}
Void display_rno()
{
Cout<<“Roll_no=”<<rno<<”\n”;
}
};
Class test: Public stud
{
Protected:
Int sub1,sub2;
Public:
Void get_mark(int m1,int m2)
{
Sub1=m1;
Sub2=m2;
}
Void display_mark()
{
Cout<<”sub1”<<sub1<<”\n”; cout<<”sub2”<<sub2<<”\n”;
}
};
Class sports
{
Protected:
Float score;
Public :
Void get_score(float s)
{
    Score=s;
}
Void put_score()
{
Cout<<”Score :”<<score<<”\n”;
}
};
Class result: public test ,public sports
{
float total;
Public:
Void display();
};
Void result::display()
{
Total=sub1+sub2+score;
display_rno();
display_mark();
put_score();
cout<<” total score:”<<total<<”\n”;
}
int main()
{
  Result r1;
  r1.getno(123);
  r1.get_mark(60,80)
  r1.get_score(6);
  r1.display();
}

- Private Inheritance

Consider the following classes:

class A {
  /*......*/;
};

Effect of Inheritance
class C: private A
{ /*
   .
   .
   .
   */
}

All the public parts of class A and all the protected parts of class A, become private members/parts of the derived class C in class C. No private member of class A can be accessed by class C. To do so, you need to write public or private functions in the Base class. A public function can be accessed by any object, however, private function can be used only within the class hierarchy that is class A and class C and friends of these classes in the above cases.

- **Public Inheritance**
  Consider the following classes:
  class A{/*........*/};
  class E: public A
  { /*
  .
  .
  .
  */
  }
  Now, all the public parts of class A become public in class E and protected part of A become protected in E.

- **Protected Inheritance**
  Consider the following classes:
  class E: protected A
  { /*
  .
  .
  .
  */
  }
  Now, all the public and protected parts of class A become protected in class E.
  No private member of class A can be accessed by class E.
  Let us take a single example to demonstrate the inheritance of public and private type in more details.
  Let us assume a class close_shape as follows:
  class closed_shape
  {
  public:
  .
  .
  .
  }
  class circle: public closed_shape
  // circle is derived in public access mode from class
  // closed-shape
float x, y; // Co-ordinates of the centre of the circle
float radius;
public:
  
  
  
  
}
class semi-circle : public circle
{
private:
  
  
  public:
  
  
  
}
class rectangle: private closed_shape
{
  float x y ;
  public:
    
    
    
    
};
class rounded_rectangle : public rectangle
{
private:
public :
  
  
  
}

**Virtual Base Class**
Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical instances are involved. Consider the following example illustrated in a figure.

- The child has two direct base classes fees and academics which themselves have a common base class ‘school’.
- The child inherits the traits of ‘school’ via two separate paths.
- It can also inherit directly as shown by the broken line.
- The ‘school’ is sometimes referred to as indirect base class. All the public and protected members of ‘school’ are inherited into ‘child’ twice, first via ‘fees’ and again via ‘academics’.
- This means, ‘child’ would have duplicate sets of the members inherited from ‘school’.
- This 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 class by declaring the base class as shown below:

```cpp
class school
{------------------}
};
class fees: virtual public school
{------------------}
};
class academics: public virtual school
{------------------}
};
class child: public fees, public academics
{
// only one copy of school will be inherited.
};
```

**Polymorphism, virtual functions, pure virtual functions:**

- **Polymorphism**
  - **Compile-time**
  - **Function**
  - **Operator**
  - **Run-time**
  - **Virtual**

**Polymorphism**

- Polymorphism is the ability of an object to take on many forms.
- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
For example, given a base class *shape*, polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as circles, rectangles and triangles.

It has two distinct aspects:
- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.
- Base classes may define and implement virtual methods, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual functions:
- Polymorphism is a mechanism that enables same interface functions to work with the whole class hierarchy.
- Polymorphism mechanism is supported in C++ by the use of virtual functions.
- The concept of virtual function is related to the concept of dynamic binding.
- The term Binding refers to binding of actual code to a function call.
Dynamic binding also called late binding is a binding mechanism in which the actual function call is bound at run-time and it is dependent on the contents of function pointer at run time.

It meant that by altering the content of function pointers, we may be able to call different functions having a same name but different code, that is demonstrating polymorphic behaviour.

If there are member functions with same name in base class and derived class, virtual functions gives programmer capability to call member function of different class by a same function call depending upon different context.

This feature in C++ programming is known as polymorphism which is one of the important features of OOP.

If a base class and derived class has same function and if you write code to access that function using pointer of base class then, the function in the base class is executed even if, the object of derived class is referenced with that pointer variable.

This can be demonstrated by an example.

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

class B
{
    public:
        void display()
        {
            cout<<"Content of base class.\n";
        }
};

class D : public B
{
    public:
        void display()
        {
            cout<<"Content of derived class.\n";
        }
};

int main()
{
    B *b;
    D d;
    b->display();

    b = &d; // Address of object d in pointer variable
    b->display();
    return 0;
}

ANOTHER EXAMPLE: Let us look into an example for the above concept:
```
```cpp
#include <iostream.h>
class employee
{
    public:
    char *name;
    char *department;
    employee (char *n, char *d)
    {
        name = n;
        department = d;
    }
    virtual void print ( )
    {};
    void employee:: print ( )
    {
        cout << "name:" << name;
        cout << "department:" << department;
    }
    class manager : public employee
    {
        public:
        short position;
        manager (char *n, char *d, short p) : employee (n, d)
        {
            name = n;
            department = d;
            position = p;
        }
        void print ( )
        {
            cout << name << "\n" << department << "\n" << position;
        }
    }
    void main ( )
    {
        employee* e ("john", "sales");
        manager* m ("james", "marketing", 3);
        e print ( )
        m print ( )
    }
}

STATIC POLYMORPHISM OR COMPILE TIME POLYMORPHISM
- It means existence of an entity in various physical forms simultaneously.
  Static polymorphism refers to the binding of functions on the basis of their signature
  (number, type and sequence of parameters).
- It is also called **early binding** because the calls are type and sequence of parameters).
- It is also called **early binding** because the calls are already bound to the proper type of
  functions during the compilation of the program.
- For example,

```
void volume (int); //prototype
void volume (int,int,int); //prototype

- When the function volume () is invoked, the passed parameters determine which one to be executed. This resolution takes place at compile time.

DYNAMIC POLYMORPHISM

- It means change of form by entity depending on the situation.
- A function is said to exhibit dynamic polymorphism if it exists in various forms, and the resolution to different function calls are made dynamically during execution time.
- This feature makes the program more flexible as a function can be called, depending on the context.

STATIC AND DYNAMIC BINDING

- As stated earlier the dynamic binding is more flexible, and the static binding is more efficient in certain cases.
- Statically bound functions do not require run-time search, while the dynamic function calls need it.
- But in case of dynamic binding, the function calls are resolved at execution time and the user has the flexibility to alter the call without modifying the source code.
- For a programmer, efficiency and performance are more important, but to the user, flexibility and maintainability are of primary concern.
- So a trade-off between the efficiency and flexibility can be made.