Unit III

Abstract class

An abstract class is a class that is designed to be specifically used as base class. An abstract class contains at least one pure virtual functions.

We can declare pure virtual function by using a pure specifier (=0) in the declaration of a virtual member function in the class declaration.

Ex:-
```
class AB
{
public:
  virtual void f() = 0;
};
```

→ function AB::f is a pure virtual function. A function declaration cannot have both pure specifier and a definition.

Ex:-
```
class AB
{
public:
  virtual void f() { y = 0; }
};
```
] this is invalid

→ You cannot use abstract class as parameter type, a function return type, nor can declare an object of an abstract class.

→ You can declare pointers and references to an abstract class.

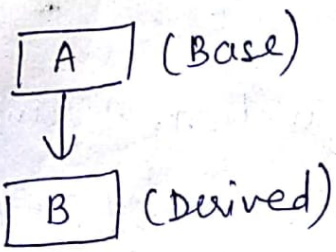
→ Virtual member functions are inherited. A class derived from an abstract ~~class~~ base class will also be abstract unless you override each pure virtual function in derived class.

Inheritance

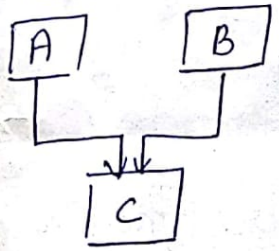
- Process of creating new classes called derived class- sub classes or child class) from base class.
- Existing classes are often called base class or super class.
- Derived class inherits all capabilities of base class and can also add new functionalities.
- Inheritance supports the concept of reusability.

Type of Inheritance

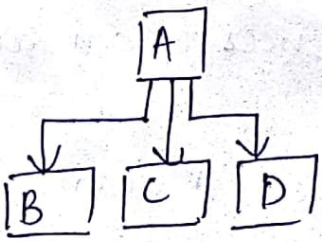
① **Single Inheritance**: A derived class with only one base class.



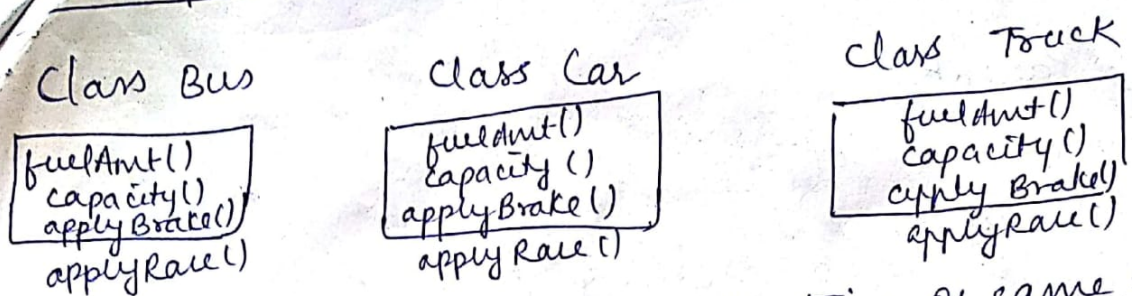
② **Multiple Inheritance** : A derived class with several base classes.



③ **Hierarchical Inheritance** : The traits of one class may be inherited by more than one class. This process is called as hierrarchical inheritance.

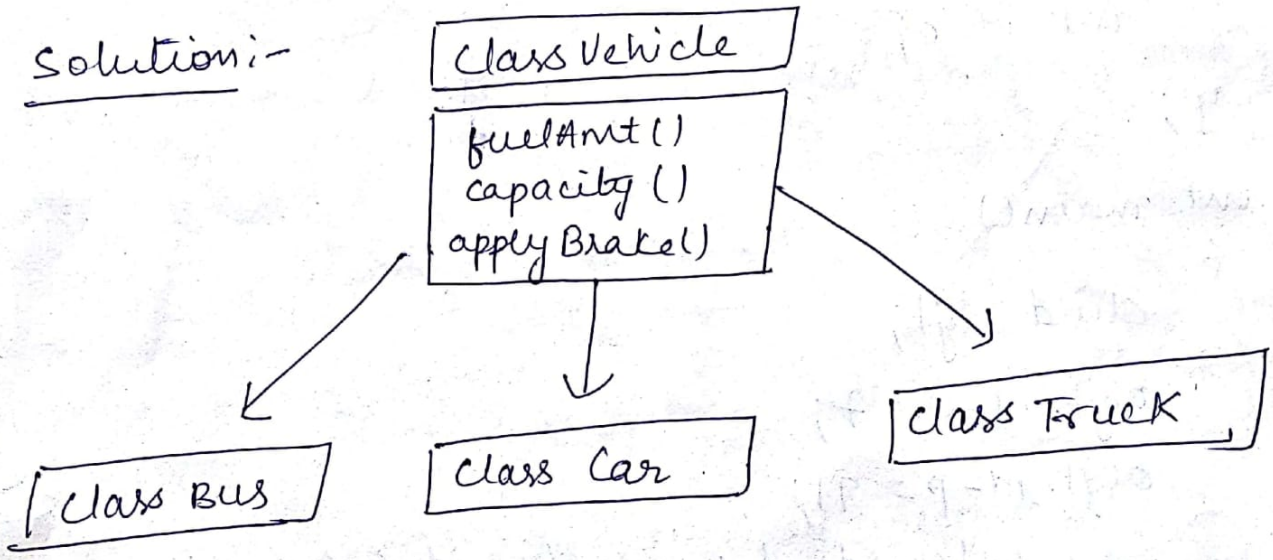


Example



Above process results in duplication of same code 3 times. This increases the chance of error and data redundancy.

Solution:-



The capability of a class to drive properties and characteristics from another class is called Inheritance

Sub class:- The class that inherits properties from another class is called so. (or Derived class)

Super class: The class whose properties are inherited by sub-class is called Base class or Super class.

Ex:

```
class Parent
{
    public:
    int id_p;
};
class child : public Parent
{
    public:
    int id_c;
};
int main()
{
    child obj1;
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "child id is " << obj1.id_c << endl;
    cout << "Parent id is " << obj1.id_p << endl;
    return 0;
}
```

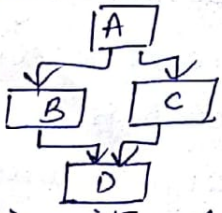
O/P:

```
child id is 7
Parent id is 91
```

// Child class is publicly inherited from Parent class so the public data members of class Parent will also be inherited by class 'child'.

Multilevel inheritance : Mechanism of deriving a class from another 'derived' class is known as multilevel inheritance.

e) **Hybrid**



Syntax : (Derived class)

A derived class can be defined by specifying its relationship with base class in addition to its own details.

```
class derived-class : visibility-mode baseclass-name
{
    . . . //
    . . . // members of derived class
}; . . . //
```

→ The colon indicates that derived class is derived from base class.

→ Visibility mode is optional and if present may be either private or public. Default is private. Visibility mode specifies whether the features of base class are privately derived or publicly derived.

Example

```
class derived : private base    // private derivation
{
  members of derived
};
```

• When base class is privately inherited, 'the public members' of base class become 'private members' of derived class and therefore the public members of base class can only be accessed by member functions of derived class.
→ they are inaccessible to the objects of derived class.
• (Remember, public member of class can be accessed by its own objects using dot operator).

Ex 2-

```
class derived : public base   // public derivation
{
  members of derived
};
```

(2) When the base class is publicly inherited, 'public members' of base class become 'public members' of derived class and therefore accessible to the objects of derived class.

* In both the cases, private members are not inherited and therefore, the private members of base class will never become the members of its derived class.

* In inheritance, some of the base class elements and member functions are inherited into derived class. We can add our own data and member functions & thus extend the functionality of base class.

Visibility of inherited members

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private →	Not inherited	Not Inherited	Not inherited
Protected →	Protected	Private	Protected
Public →	Public	Private	Protected

Access specifier	Accessible from own class	Accessible from derived class	Accessible from objects outside class
Public	✓	✓	✓
Private	✓	X	X
Protected	✓	✓	X

Single Inheritance (Publicly derived)

```
class student
{
public:
int role;
char name[20];
void get()
{
cout << "Roll:";
cin >> role;
cout << "name:";
cin >> name;
};
```

• When base class is privately inherited ...

```
class marks : public student
{
  int s1, s2, s3, total;
   float per;
   public :
   void get_m()
   {
    cout << " marks :";
    cin >> s1 >> s2 >> s3;
   }
   void calculate()
   {
    total = s1+s2+s3;
    per = total * 100.0/300;
   }
   void display()
    {
      cout << roll << total << name << per;
   }
   void main()
   {
     marks m;
     m.get();
     m.get_m();
     m.calculate();
     m.display();
  }
```

Inheritance

Accessibility

```
class base
{
  public:
   int x;
  protected:
   int y;
  private:
   int z;
};

class pDerived : public Base
{ // x is public
  // y protected
  // z is not accessible
};

class proDerived : protected base
{ // x protected
  // y protected
  // z is not accessible
};

class priDerived : private base
{ //x is private
  // y private
  // z is not accessible
};
```

Program: Single Inheritance

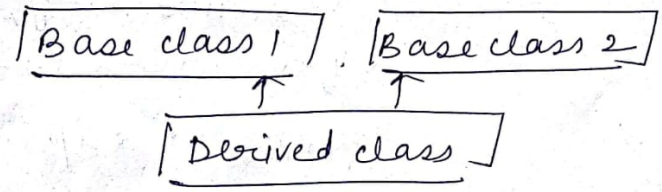
```
class Shape
{
protected:
int width, height;
public:
void setWidth (int w)
 { width = w;
 }
void setHeight (int h)
 { height = h;
 }
};
class Rectangle : public Shape
 { public:
    int getArea()
    { return (width * height);
    }
};
int main(void)
{
  Rectangle R;
  R.setWidth (15);
  R.setHeight (10);
  cout << " Area :" << Rect.getArea() << endl;
  return 0;
}
```

O/P

Area : 150.

Multiple Inheritance

It is the inheritance where a class can inherit from more than one classes.




```
Eg:  class A
     {  public:
        void print()
        {
          cout << " class A printing " << endl;
        }
     };
     class B
     {
        public:
        void print()
        {
          cout << " In class B " << endl;
        }
     };
     Class AB : public A, public B
     {
        public:
        void print ~~abc~~ () // overriding
        {
          cout << " In class AB " << endl;
        }
     };
     int main()
     {
        AB obj;
        obj.print();
     }
```

O/P

In class AB

(2)

```
class A
{
protected:
    int x;
public:
void get_x();
};
class B
{
protected:
    int y;
public:
 void get_y();
};
class C : public A, public B
{
    int sum;
public:
    void add(void);
    void disp(void);
};
void A::get_x(void)
{
    cout<<"Enter x:";
    cin>>x;
}
void B::get_y(void)
{
cout<<"Enter y:";
cin>>y;
}
void C::add(void)
{ sum = x+y;
}
void C::disp(void)
{
cout<<"value of x:"<<x<<endl;
cout<<"value of y:"<<y<<endl;
cout<<"sum:"<<sum<<endl;
}
int main()
{
    C ob;
 ob.get_x();
 ob.get_y();
 ob.add();
 ob.disp();
return 0;
}
```

O/P :

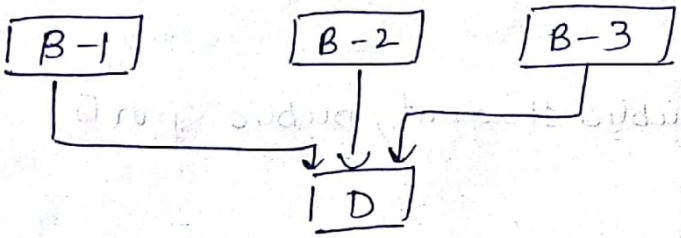
```
Enter value of X : 10
      "        Y : 20
Value of X : 10
Value of Y : 20
Sum : 30
```

C++ users in americ...

Meng

Multiple Inheritance

A class inherit the attributes of two or more classes. This is known as multiple inheritance. Multiple inheritance allows 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 intelligence of another.



Syntax for derived :-

```
class D: visibility B-1, visibility B-2, ...
{
    // Body of D
};
```

Ex. class student
{
protected:
int rno, m1, m2;
public:
void get()
{ cout << "Enter Roll:";
cin >> rno;
cout << "Enter 2 marks";
cin >> m1 >> m2;
};
};

```
class sports
{
  protected:
   int sm;
  public:
  void getsm()
  {
     cout << "Enter sports marks:";
     cin >> sm;
  }
};

class statement : public student, public sports
{
   int tot, avg;
   public:
   void display()
   {
      tot = (m1 + m2 + m3 + sm);
      avg = tot/3;
      cout << "Roll no : " << rno << "\n Total : " << tot;
      cout << "Avg : " << avg;
   }
};
Void main()
{
  clrscr();
  statement obj;
  obj.get();
  obj.getsm();
  obj.display();
  getch();
}
```

O/P

```
Enter Roll : 100
Enter 2 marks
  90
  80
Enter sport mark : 90
 Roll : 100
 Total : 260
 Avg : 86.66
```

Ex (2)

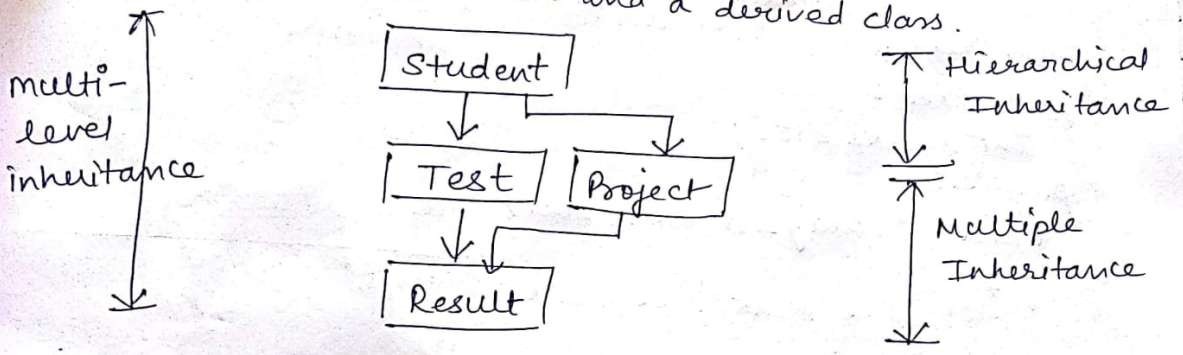
```
class M
{
    protected:
        int m;
    public:
        void get_m(int);
};

class N
{
    protected:
        int n;
    public:
        void get_n(int);
};

class P : public M, public N
{
    public:
        void display(void);
};

void M::get_m(int x)
{
    m = x;
}

void N::get_n(int y)
{
    n = y;
}

void P::display(void)
{
    cout << "m = " << m << "\n";
    cout << "n = " << n << "\n";
    cout << "m * n " << m * n << "\n";
}

int main()
{
    P p;
    p.get_m(10);
    p.get_n(20);
    p.display();
    return 0;
}
```

O/P

```
m = 10
n = 20
m * n = 200
```

Virtual Base class

When a class is made a virtual base class, C++ taces necessary care to see that only one copy of that class is inherited, regardless of how many inheritance path exist between virtual base class and a derived class.



Result class receives the data of student twice, one through Test intermediate class, and secondly, through project class (intermediate). So, to avoid the redundancy or ambiguity, we declare or introduce the concept of virtual Base class. We declare student as virtual. This results in producing only one single copy of student to Result class.

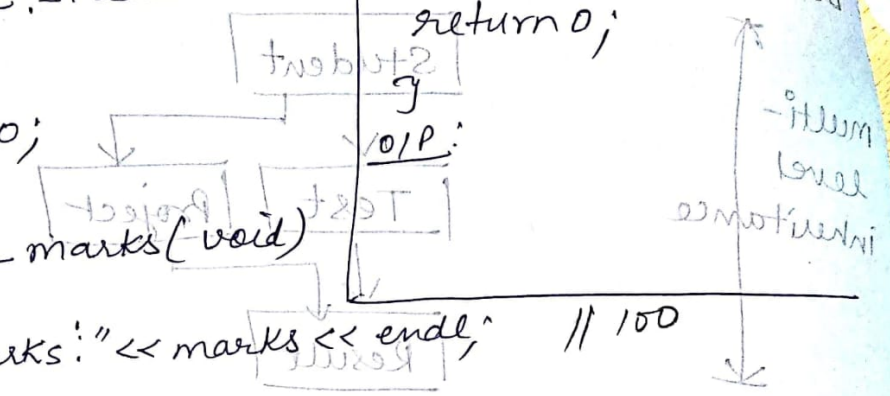
Declaration

```
class student
{
};
class Test : virtual public student
{
};
class Project : virtual public student
{
};
class Result : public Test, public Project
{
};
// only 1 copy of class student will be inherited
```

Scanned by CamScanner

Code:

```
class student
{
  protected:
    char name[50] = "Abc";
};
class Test : virtual public student
{
  protected:
    int marks = 100;
  public:
    void display_marks(void)
    {
      cout << " Marks : " << marks << endl;   // 100
    }
};
class Project : virtual public student
{
  protected:
    int p_marks = 200;
  public:
    void display_pmarks(void)
    {
      cout << " Project marks : " << p_mark << endl;  // 200
    }
};
class Result : public Test, public Project
{
  public:
    void display_Res(void)
    {
      cout << " Name : " << name << endl;  // Abc
      cout << " TOTal marks : " << marks + p_marks << endl;
                                                  // 300
    }
};
int main()
{
  Result res;
  res.display_marks();
  res.display_pmarks();
  res.display_Res();
  return 0;
}
```



Aggregation :- There is a single owner of the child object. These child owners cannot belong to any other owner.
- These child objects which are inside the owner can exist independently. Or other words, the workers objects can exist independently
He (Manager) has many workers under him

Composition :- The lifetime of both the objects are same. In other words, if the project goes off, the manager will have problem.
Eg - Manager's salary depends on project sucess.
- Project sucess depends on manager.

Abstract class

```
class Base
{ public:
   virtual void print() = 0;
};
class derived : public Base
{
  void print()
  { cout << "Inside derived class";
  }
};
int main()
{ D obj;
  obj.print();
  getch();
  return 0;
}
```

O/P
Inside derived class

GOOD WRITE

II way

```
int main()
{ // B obj ; // error
  B *ptr;
  D obj;
  ptr = &obj;
  ptr -> disp(); print();
  getch();
  return 0;
}
```

O/P

In derived class

Virtual function (or Runtime polymorphism)

```
class one          // Parent class
{ public:
  void show()
  { cout << "Class one show function";
  }
};
class two : public one // child class
{ public:
  void show()
  { cout << "Class Two show function";
  }
};
int main()
{ one obj;
  Two ob2;
  One *p;
  p = &ob2;
  p -> show();
}
};
```

Pointer of Parent class can store address of child class

O/P

Class Two show function

p = &ob1; → class one show funct

DATE: __/__/____
PAGE _____

Compiler in both cases looking / printing cout statements of parent class. because compiler looking for the 'data type' of pointer ("one") So, in both the cases, we are having at this point of data type 'one'. So, in every case it is making call for class one show function.

Sol^n is :→ to add keyword 'virtual'.
Virtual actualy allow programmer to call it function with the help of pointer of parent class by storing address of different objects.
So, (p = & ob2), ob2 is checked, rather than data type of pointer.

```
class one
{ public:
  virtual void show()
  { cout<<"class one";
  }};
class two : public one
{ public:
  void show()
  { cout<<"class two";
  }
};
int main()
{ one obj;
  two ob2;
  one *p;
  p = &ob2;
  p->show();
}
```

O/P :-
class two

With virtual

```
class B
{ public :
Virtual void Func()
  {
    cout << "Base class func" << endl;
  }
};
class D : public B
{
 public : void Func()
 {
 cout << " Derived class func" << endl;
 }
};
void main()
 {
    D d;
    B *ptr = &d;
    ptr -> Func();      // calls Derived class function
}
```

[with virtual keyword Pointer now Points to appropriate func" where it points]

Virtual function (Runtime Polymorphism)

Without Virtual

```
class B
{
  public:
 void func()
 {
 cout<<" Base class func"<< endl;
 };
 class D : public B
{
  public : void Func()
  {
  cout<<" Derived class func"<<endl;
  }
};
void main()
{
    D d;
    B *ptr = &d;
   ptr -> func();    // Calls Base class function
}
```

- Base class func

In this w/o virtual keyword pointer always point to base class func" irrespective of pointer holds address of derived object

as Standard approach ... templatized classes and Me
... and functions (algorith...
coll...

Unit III

Polymorphism

- means many forms.
  Poly :- means Many
  Morph :- means form

- It occurs when there is hierarchy of classes and they are related by inheritance
- C++ polymorphism means that call to a member function will cause different functions to be executed depending on the type of object that invokes the function.
- Polymorphism means more than one function with same name but with different working.

Type of Polymorphism

Compile-time / Static Polymorphism / Early Binding	Runtime / Dynamic Polymorphism / Late Binding
Eg. function Overloading, Operator overloading	Eg Virtual function

function Overloading :-
More than one function with same name with different signature in a class or in a same scope is called function overloading.

operator overloading :-
Way of providing new implementation of existing operators to work with user-defined data types.

Virtual function :→ It is used when we need to invoke derived class function using base class pointer. Giving new implementation of derived class method into base class and calling of this new implementation of function

~~derived class method into base class and the calling~~ with base class's object is done by making base class as virtual function. this is how, can achieve "Run Polymorphism".

Scanned by CamScanner

Polymorphism in C++

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee. So a same person posses have different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

1. **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.
   - **Function Overloading**: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

   Rules of Function Overloading

```
// C++ program for function overloading

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:

    // function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }

    // function with same name but 1 double para
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // function with same name and 2 int paramet
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << "
    }
};

int main() {

    Geeks obj1;

    // Which function is called will depend on t
    // The first 'func' is called
    obj1.func(7);

    // The second 'func' is called
    obj1.func(9.132);

    // The third 'func' is called
    obj1.func(85,64);
    return 0;
}
```

Run on IDE

Output:

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

In the above example, a single function named *func* acts differently in three different situations which is the property of polymorphism.

- **Operator Overloading:** C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add to operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.

Example:

```
// CPP program to illustrate
// Operator Overloading
#include<iostream>
using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i) y?

    // This is automatically called when '+' is
    // between two Complex objects
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;

    }
    void print() { cout << real << " + i" << ima;
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to
    c3.print();
}
```

Run on IDE

Output:

```
12 + i9
```

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers. To learn operator overloading in details visit thislink.

2. **Runtime polymorphism:** This type of polymorphism is achieved by Function Overriding.

are derived

- **Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// C++ program for function overriding

#include <bits/stdc++.h>
using namespace std;

// Base class
class Parent
{
    public:
    void print()
    {
        cout << "The Parent print function was c
    }
};

// Derived class
class Child : public Parent
{
    public:

    // definition of a member function already p
    void print()
    {
        cout << "The child print function was ca
    }

};

//main function
int main()
{
    //object of parent class
    Parent obj1;

    //object of child class
    Child obj2 = Child();


    // obj1 will call the print function in Pare
    obj1.print();

    // obj2 will override the print function in
    // and call the print function in Child
    obj2.print();
    return 0;
}
```

Run on IDE

Output:

```
The Parent print function was called
The child print function was called
```

POLYMORPHISM IN C++

he word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are elated by inheritance.

++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object iat invokes the function.

onsider the following example where a base class has been derived by other two classes –

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

class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};

// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;

   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;

   // call triangle area.
   shape->area();

   return 0;
}
```

/hen the above code is compiled and executed, it produces the following result –

The reason for the incorrect output is that the call of the function area is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area in the Shape class with the keyword **virtual** so that it looks like this –

```
class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result –

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area function is called.

As you can see, each of the child classes has a separate implementation for the function area. This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which i is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area in the base class to the following –

```
class Shape {
   protected:
      int width, height;

   public:
      Shape(int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      // pure virtual function
      virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

Operator Overloading in C++

In C++, we can make operators to work for user defined classes. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.
Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

A simple and complete example

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

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Run on IDE

Output:

```
12 + i9
```

What is the difference between operator functions and normal functions?

Operator functions are same as normal functions. The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.
Following is an example of global operator function.

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

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}
    void print() { cout << real << " + i" << imag << endl; }

// The global operator function is made friend of this class so
// that it can access private members
friend Complex operator + (Complex const &, Complex const &);
};


Complex operator + (Complex const &c1, Complex const &c2)
{
     return Complex(c1.real + c2.real, c1.imag + c2.imag);
}


int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
    return 0;
}
```

Run on IDE

Can we overload all operators?

Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.

. (dot)
::
?:
sizeof

Why can't . (dot), ::, ?: and sizeof be overloaded?
See this for answers from Stroustrup himself.

Important points about operator overloading
1) For operator overloading to work, at leas one of the operands must be a user defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor). See this for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

```
#include <iostream>
using namespace std;
class Fraction
{
    int num, den;
public:
    Fraction(int n,  int d) { num = n; den = d; }

    // conversion operator: return float value of fraction
    operator float() const {
        return float(num) / float(den);
    }
};

int main() {
    Fraction f(2, 5);
    float val = f;
    cout << val;
    return 0;
}
```

Run on IDE

Output:

```
0.4
```

Overloaded conversion operators must be a member method. Other operators can either be member method or global method.

4) Any constructor that can be called with a single argument works as a conversion constructor, means it can also be used for implicit conversion to the class being constructed.

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

class Point
{
private:
    int x, y;
public:
    Point(int i = 0, int j = 0) {
        x = i;   y = j;
    }
    void print() {
        cout << endl << " x = " << x << ", y = " << y;
    }
};

int main() {
    Point t(20, 20);
    t.print();
    t = 30;   // Member x of t becomes 30
    t.print();
    return 0;
}
```

O/P
x=20, y=20
x=30, y=0

Exception Handling

most common types

Unit IV

Operator Overloading

Operator is a symbol that tells the compiler to perform specific task. Every operator have their own functionality to work with in data types. Class is user-defined data type and compiler doesn't understand, how to use operators with user-defined types. To use operators with user-defined data types, they need to be overload according to a programmer's requirement.
Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types.
An operator can be overloaded by defining a function to it. The function for operator is declared by using the **operator** key followed by the operator.
There are two types of operator overloading in C++

Binary Operator Overloading
Unary Operator Overloading

Overloading Binary Operator

Binary operator is an operator that takes two operand(variable). Binary operator overloading is similar to unary operator overloading except that a binary operator overloading requires an additional parameter.

Binary Operators

Arithmetic operators (+, -, *, /, %)
Arithmetic assignment operators (+=, -=, *=, /=, %=)
Relational operators (>, <, >=, <=, !=, ==)

Example of Binary Operator Overloading

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

class Rectangle
{

    int L,B;

    public:

    Rectangle()         //Default Constructor
    {
        L = 0;
        B = 0;
    }

    Rectangle(int x,int y)         //Parameterize Constructor
    {
        L = x;
        B = y;
    }

    Rectangle operator+(Rectangle Rec)     //Binary operator overloading func.
    {
        Rectangle R;

            R.L = L + Rec.L;
            R.B = B + Rec.B;

        return R;
    }

    void Display()
    {
        cout<<"\n\tLength : "<<L;
        cout<<"\n\tBreadth : "<<B;
```

```
};
void main()
{
    Rectangle R1(2,5),R2(3,4),R3;
    //Creating Objects

    cout<<"\n\tRectangle 1 : ";
    R1.Display();

    cout<<"\n\n\tRectangle 2 : ";
    R2.Display();

    R3 = R1 + R2;        Statement 1

    cout<<"\n\n\tRectangle 3 : ";
    R3.Display();
}
```

Output :

```
Rectangle 1 :
L : 2
B : 5

Rectangle 2 :
L : 3
B : 4

Rectangle 3 :
L : 5
B : 9
```

In statement 1, Left object R1 will invoke operator+() function and right object R2 is passing as argument.
Another way of calling binary operator overloading function is to call like a normal member function as follows,

```
R3 = R1.operator+ ( R2 );
```

Overloading Unary Operator

Unary operator is an operator that takes single operand(variable). Both increment(++) and decrement(--) operators are u operators.

Example of Unary Operator Overloading

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

class Rectangle
{
```

```
int L,B;

public:

Rectangle()          //Default Constructor
{
    L = 0;
    B = 0;
}

void operator++()    Unary operator overloading func.
{
    L+=2;
    B+=2;
}

void Display()
{
    cout<<"\n\tLength : "<<L;
    cout<<"\n\tBreadth : "<<B;
}

};


void main()
{

    Rectangle R;
    //Creating Object


    cout<<"\n\tLength Breadth before increment";
    R.Display();


    R++;

    cout<<"\n\n\tLength Breadth after increment";
    R.Display();
}
```

Output :

```
Length Breadth after increment
L : 0
B : 0

Length Breadth after increment
L : 2
B : 2
```

Function Overloading in C++

More than one function with same name, with different signature in a class or in a same scope is called function overloading

function includes :
Number of arguments
Type of arguments
Sequence of arguments

When you call an overloaded function, the compiler determines the most appropriate definition to use by comparing the signature of ca
statement with the signature specified in the definitions.

Example of function overloading

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

class CalculateArea
{

    public:
    void Area(int r)        //Overloaded Function 1
    {
        cout<<"\n\tArea of Circle is : "<<3.14*r*r;
    }

    void Area(int l,int b)        //Overloaded Function 2
    {
        cout<<"\n\tArea of Rectangle is : "<<l*b;
    }

    void Area(float l,int b)        //Overloaded Function 3
    {
        cout<<"\n\tArea of Rectangle is : "<<l*b;
    }

    void Area(int l,float b)        //Overloaded Function 4
    {
        cout<<"\n\tArea of Rectangle is : "<<l*b;
    }

};


void main()
{

    CalculateArea C;

    C.Area(5);      //Statement 1
    C.Area(5,3);    //Statement 2
    C.Area(7,2.1f);  //Statement 3
    C.Area(4.7f,2);  //Statement 4

}
```

Output :

```
Area of Circle is : 78.5
Area of Rectangle is : 15
Area of Rectangle is : 14.7
Area of Rectangle is : 29.4
```

In the above example, we have four member functions named Area. Statement 1 will invoke the function 1 b'coz the signature of function 1 is similar to the statement 1. Similarly Statement 3 will invoke function 4 b'coz statement 3 is passing two arguments, 1st is of integer type and 2nd is of float type. Function 4 is the only function who is receiving integer and float respectively.

Operator Overloading

In C++, it's possible to change the way operator works (for user-defined types). In this article, you will learn to implement operator overloading feature.

```
ClassName operator - (ClassName c2)
{
   ... .. ...
   return result;
}

int main()
{
   ClassName c1, c2, result;
   ... .. ....
   result = c1-c2;
   .... .. ...
}
```

The meaning of an operator is always same for variable of basic types like: int, float, double etc. For example: To add two integers, + operator is used.

However, for user-defined types (like: objects), you can redefine the way operator works. For example:

If there are two objects of a class that contains string as its data members. You can redefine the meaning of + operator and use it to concatenate those strings.

This feature in C++ programming that allows programmer to redefine the meaning of an operator (when they operate on class objects) is known as operator overloading.

Why is operator overloading used?

You can write any C++ program without the knowledge of operator overloading. However, operator operating are profoundly used by programmers to make program intuitive. For example,

You can replace the code like:

```
calculation = add(multiply(a, b),divide(a, b));
```

to

```
calculation = (a*b)+(a/b);
```

```
calculation = add(multiply(a, b),divide(a, b));
```

to

```
calculation = (a*b)+(a/b);
```

How to overload operators in C++ programming?

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.

Example: Operator overloading in C++ Programming

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

class Test
{
    private:
        int count;

    public:
         Test(): count(5){}

         void operator ++()
         {
             count = count+1;
         }
         void Display() { cout<<"Count: "<<count; }
};

int main()
{
     Test t;
     // this calls "function void operator ++()" function
     ++t;
     t.Display();
     return 0;
}
```

Exception Handling Unit IV

* two nr
syntactic
Logical
and

This function is called when ++ operator operates on the object of Test class (object t in this case).

In the program, void operator ++ () operator function is defined (inside Test class).

This function increments the value of count by 1 for t object.

Things to remember

1. Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.).
2. Two operators = and & are already overloaded by default in C++. For example: To copy objects of same class, you can directly use = operator. You do not need to create an operator function.
3. Operator overloading cannot change the precedence and associatively of operators. However, if you want to change the order of evaluation, parenthesis should be used.
4. There are 4 operators that cannot be overloaded in C++. They are :: (scope resolution), . (member selection), .* (member selection through pointer to function) and ?: (ternary operator).

Following best practices while using operator overloading

Operator overloading allows you to define the way operator works (the way you want).

In the above example, ++ operator operates on object to increase the value of data member count by 1.

```
void operator ++()
    {
        count = count+1;
    }
```

However, if you use the following code. It decreases the value of count by 100 when ++ operator is used.

```
void operator ++()
    {
        count = count-100;
    }
```

This may be technically correct. But, this code is confusing and, difficult to understand and debug.

It's your job as a programmer to use operator overloading properly and in consistent way.

In the above example, the value of count increases by 1 when ++ operator is used. However, this program is incomplete in sense that you cannot use code like:

```
t1 = ++t
```

It is because the return type of the operator function is void .

verloading stream insertion (<>) operators in C++

In C++, stream insertion operator "<<" is used for output and extraction operator ">>" is used for input.

We must know following things before we start overloading these operators.

1) cout is an object of ostream class and cin is an object istream class

2) These operators must be overloaded as a global function. And if we want to allow them to access private data members of class, we must make them friend.

Why these operators must be overloaded as global?

In operator overloading, if an operator is overloaded as member, then it must be a member of the object on left side of the operator. For example, consider the statement "ob1 + ob2" (let ob1 and ob2 be objects of two different classes). To make this statement compile, we must overload '+' in class of 'ob1' or make '+' a global function.

The operators '<<' and '<<' are called like 'cout << ob1' and 'cin << ob1'. So if we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time. Therefore, these operators are overloaded as global functions with two parameters, cout and object of user defined class.

Following is complete C++ program to demonstrate overloading of <> operators.

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

class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {  real = r;   imag = i; }
    friend ostream & operator << (ostream &out, const Com
    friend istream & operator >> (istream &in,  Complex &
};

ostream & operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}

istream & operator >> (istream &in,  Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imagenory Part ";
    in >> c.imag;
    return in;
}

int main()
{
   Complex c1;
   cin >> c1;
   cout << "The complex object is ";
   cout << c1;
   return 0;
}
```

Run on IDE

Output:

```
Enter Real Part 10
Enter Imagenory Part 20
The complex object is 10+i20
```

Adding two distances using binary plus (+) operator overloadi

in c++.

```
/*C++ program to add two distances using binary plus (+) operator

#include<iostream>
using namespace std;

class Distance
{
    private:
        int feet,inches;

    public:
        //function to read distance
        void readDistance(void)
        {
            cout << "Enter feet: ";
            cin >>feet;
            cout << "Enter inches: ";
            cin >>inches;
        }
        //function to display distance
        void dispDistance(void)
        {
            cout << "Feet:" << feet << "\t" << "Inches:" << inche
        }
        //add two Distance using + operator overloading
        Distance operator+(Distance &dist1)
        {
            Distance tempD;     //to add two distances
            tempD.inches= inches + dist1.inches;
            tempD.feet  = feet   + dist1.feet + (tempD.inches/12)
            tempD.inches=tempD.inches%12;
            return tempD;
        }
};
int main()
{
    Distance D1,D2,D3;

    cout << "Enter first  distance:" << endl;
    D1.readDistance();
    cout << endl;

    cout << "Enter second distance:" << endl;
    D2.readDistance();
    cout << endl;

    //add two distances
    D3=D1+D2;

    cout << "Total Distance:" << endl;
    D3.dispDistance();

    cout << endl;
    return 0;
}
```

```
Enter first  distance:
Enter feet: 22
Enter inches: 10

Enter second distance:
Enter feet: 23
Enter inches: 11

Total Distance:
Feet:46     Inches:9
```

Unary increment (++) and decrement (--) operator overloading program in c++.

```
/*C++ program for unary increment (++) and decreme

#include<iostream>
using namespace std;

class NUM
{
    private:
        int n;

    public:
        //function to get number
        void getNum(int x)
        {
            n=x;
        }
        //function to display number
        void dispNum(void)
        {
            cout << "value of n is: " << n;
        }
        //unary ++ operator overloading
        void operator ++ (void)
        {
            n=++n;
        }
        //unary -- operator overloading
        void operator -- (void)
        {
            n=--n;
        }
};
int main()
{
    NUM num;
    num.getNum(10);

    ++num;
    cout << "After increment - ";
    num.dispNum();
    cout << endl;

    --num;
    cout << "After decrement - ";
    num.dispNum();
    cout << endl;
    return 0;
}
```

```
After increment - value of n is: 11
After decrement - value of n is: 10
```

Unary minus (-) operator overloading program in c++

```
/*C++ program for unary minus (-) operator overloading.*/

#include<iostream>
using namespace std;

class NUM
{
    private:
        int n;

    public:
        //function to get number
        void getNum(int x)
        {
            n=x;
        }
        //function to display number
        void dispNum(void)
        {
            cout << "value of n is: " << n;
        }
        //unary - operator overloading
        void operator - (void)
        {
            n=-n;
        }
};

int main()
{
    NUM num;
    num.getNum(10);
    -num;
    num.dispNum();
    cout << endl;
    return 0;

}
```

Output

```
value of n is: -10
```

Unit III

Page No.
Date : / /

Unary operator overloading: There is no need of explicit objects (require only one operand)

→ ++, --

eg: a++, a--

When binary operator be overloaded we provide one object implicit and one object explicit.

```
#include <iostream.h>
class Test
{
int a;
public:
Test()
{
  a=0;
}
void operator ++()
{
  a++;
}
void operator --()
{
  a--;
}
void show()
{
cout<<a;
}};
```

used for automatic initialization of variable

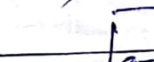
```
void main()
{
Test t;
t++;
```

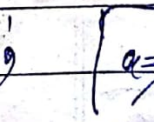
only one argument

when t++ called then a++

(++ work only on integer data but here ++, -- operator on user defined datatype)

```
t.show();
t--;
t.show();
}
```





Unit III (SS)

Page No.: (22)
Date: / /

→ Overloading Insertion and Extraction operator in C++ (both worked on predefined datatypes)

eg int a=10;
Cout << a;
cin >> a (cin derived from istream)

→ we have to use here 2 classes below cout is mentioned in ostream class

→ ∴ friend concept used → declaration should be inside class

```
#include<iostream.h>
Class stu
{
int id;
Char name[10];
Public:
friend void Operator >>
(istream &in, stu &s)   reference object
{
Cout << "enter id, name";
cin >> s.id >> s.name;
}
friend void operator << (ostream
&out, stu &s)
{
Cout << s.id;
Cout << s.name;
}
};
```

```
void main()
{
stu s;
Cout << s;   (invalid get garbage value here)
cin >> s;
cin >> s;
Cout << s;
}
```

Polymorphism in C++

Page No.
Date

Poly + morphs
many forms
→ One of oops feature
{ One entity exist in many forms }
eg:- one person in reality diff. existence at same time

→ Types of polymorphism

Compile time
↓
~~Achieved by Inheritance~~
→ Method overloading (function operator overloading)
→ Method overriding → Runtime polym.

Runtime
↓
Achieved by virtual functions

① Compile time polymorphism (static / early binding)
eg:- function overloading :- when there are multiple functions with same name but different parameters, then these fun" are said to be overloaded.

```
eg  class A
    {
    public:
    void func (int x)
    {
    cout << x;
    }
    void func (double x)
    {
    cout << x;
    }
```

```
void func (int x, int y)
{
cout << x << y;
}
};
int main()
{
A obj;
obj.func(1);
obj.func(9.13);
obj.func(85,64);
}
```

... of bugs are ...

finding of the problem

O/P → x = 7
x = 9.13
a = 85, 64

Page No.
Date

Called as Adhoc polymorphism

8) Operator overloading → C++ provides option to overload operators → when operators are overloaded to do multiple jobs
+ → for string concatenation (strings as operands)
→ addition (integers as operands)

class complex

returntype classname :: operator operatorsymbol (args)
{ // function body }

returntype Op.Symbol (args)
{ }

operator to be overloaded

→ Program to overload binary operator + to find sum of two complex No's

```
class complex
{
float real, img;
public:
complex (float, float);
complex operator + (complex);
void disp();
};
complex :: complex (float x, float y)
{ real = x
img = y }
```

```
complex complex :: operator + (complex c)
{
complex temp;
temp.real = real + c.real;
temp.img = img + c.img;
return (temp);
}
```

the count ...
(.), (.*), ...

eg :- Overloa...
is a ... operator

#inclu...
cla...
{
int ...
Public:
void ...
{
cout ...
}
void ...
{
}

};
main
{
test t1, ...
t1.g...
t...
t1 == t2 ...

We can't overload these operators

(.), (.*), (::), (sizeof)

Page No.
Date 20

eg:- Overload == operator (comparison operator) (find objects are equal or not)

is a binary operator

```
#include<iostream.h>
class Test
{
int a;
public:
void get()
{
cin>>a;
}
void operator==(Test t2)
{
if(a==t2.a)
cout<<"object equal";
else
cout<<"not equal";
}
};
main()
{
Test t1, t2;
t1.get();
t2.get();
t1==t2;
t1.compare(t2);
}
```

explicitly object for comparison outside.

t1 a

t2 a

left hand operand object is same like

most common types of bugs are logical errors ... of the problem

```
#include<iostream.h>
class complex
{ private:
    int a, b;
public:
void setdata(int x, int y)
    { a=x; b=y; }
void showdata()
    { cout<<a<<b; }
};
void main()
{
  complex c1, c2, c3;
  c1.setdata(3,4);
  c2.setdata(5,6);
  c3 = c1+c2; // error
```

want to add two complex No.

Compiler on see '+' both operands of primitive type it works, but here both operands are complex type so it give error.

```
  c3 = c1.add(c2);
```

c1 call add and pass c2 as argument here add is a member func^n

```
  c3.showdata();
}
```

becoz c3 is complex type

```
complex add(complex c)
{
  complex temp;
  temp.a = a + c.a;
  temp.b = b + c.b;
  return(temp);
}
```

c1 ka a, c2 ka a

a=8, b=10 (c3)

w we modi... we change ... No becoz ... type already ... In C++ if we ... symbol as ... keyword before

```
complex Oper...
{
  // sam...
}
```

...n call

or we ... dot Ope... c3

→ Here we ... concept ... are of nor...

Now we modify above program,

→ we change add() name to '+'

- No because for func^n name –, alpha, digits allowed not operator.

In C++, if we want to use operator symbol (available in C++) as func^n name then use operator keyword before the operator.

Complex operator + (Complex c)
{
→ // same
}

In call:

C3 = C1. operator+(C2);

Or we can call this function w/o dot operator also,

C3 = C1+C2; [C1 call + func^n C2 as argument and result return in C3]

→ Here we use '+' concept when operands are of non primitive types.

→ function & operator overloading are example of compile time polymorphism. The overloaded member func^n are selected for invoking by matching arguments, both type and number. The compiler knows this info^n at compile time & ∴ compiler is able to select appropriate func^n for particular call at compile time itself.

⇒ In runtime polymorphism an appropriate member func^n is selected while program is running. C++ support runtime polymorphism with help of virtual func^n. Dynamic binding requires use of pointers to objects and is one of powerful feature of C++.

→ Overloading of Subscript operator in OOP

[ ] → used in array

eg: int a[5];

1000 1004 1008 1010 1012

| 0 | 1 | 2 | 3 | 4 |

a[2] ⇒ *(a+2)

operand operand (both are of in built types)

if we write like

Array Obj;
cout << Obj[2];

Obj — invalid. memory class Array

obj is not inbuilt types

similar: t3 = t1 + t2;
or
t3 = t1.operator+(t2);

written as

Obj.operator[](2);

index value passed to array

```
class Array
{
Private:
    int a[10];
Public:
void insertdata(int index,
                int value)
{
    a[index] = value;
}
};
```

```
int main()
{
    Array Obj;
    int i;
    for(i=0; i<=9; i++)
    Obj.insertdata(i, 10*(i+1));
    for(i=0; i<=9; i++)
    cout << Obj[i];
}
```

0	10	
0	1	

Print — error

cout << Obj.operator[](i);

int operator [] (int index)

{

return (a[index]);

}

(i value passed as index variable)

Scanned by CamScanner