Unit-2

Constructors (ctor):- A constructor is a 'special' member function that is callid automatically when an object is created.

⇒ It is used to construct the object and assigns some values for object's data number.

⇒ It can be defined inside or outside the class definition.

⇒ If no constructor is defined explicitly, Compiler will automatically generate a default constructor.

⇒ Also used to allocate memory at run time using "new" operator in C++.

Characteristics of Constructors!-

⇒ They should be declared in the public section.

⇒ Invoked automatically when the objects are created.

⇒ They can't return values because they do not have return types.

⇒ They can't be inherited, though a derived class can call the base class constructor.

⇒ They make 'implicit calls' to the operators new and delete when memory allocation is required.

⇒ Constructors can not be virtual because when a constructor of a class is executed there is no virtual table in the memory, means there is not defined virtual pointer.

⇒ We can not refer to their addresses.

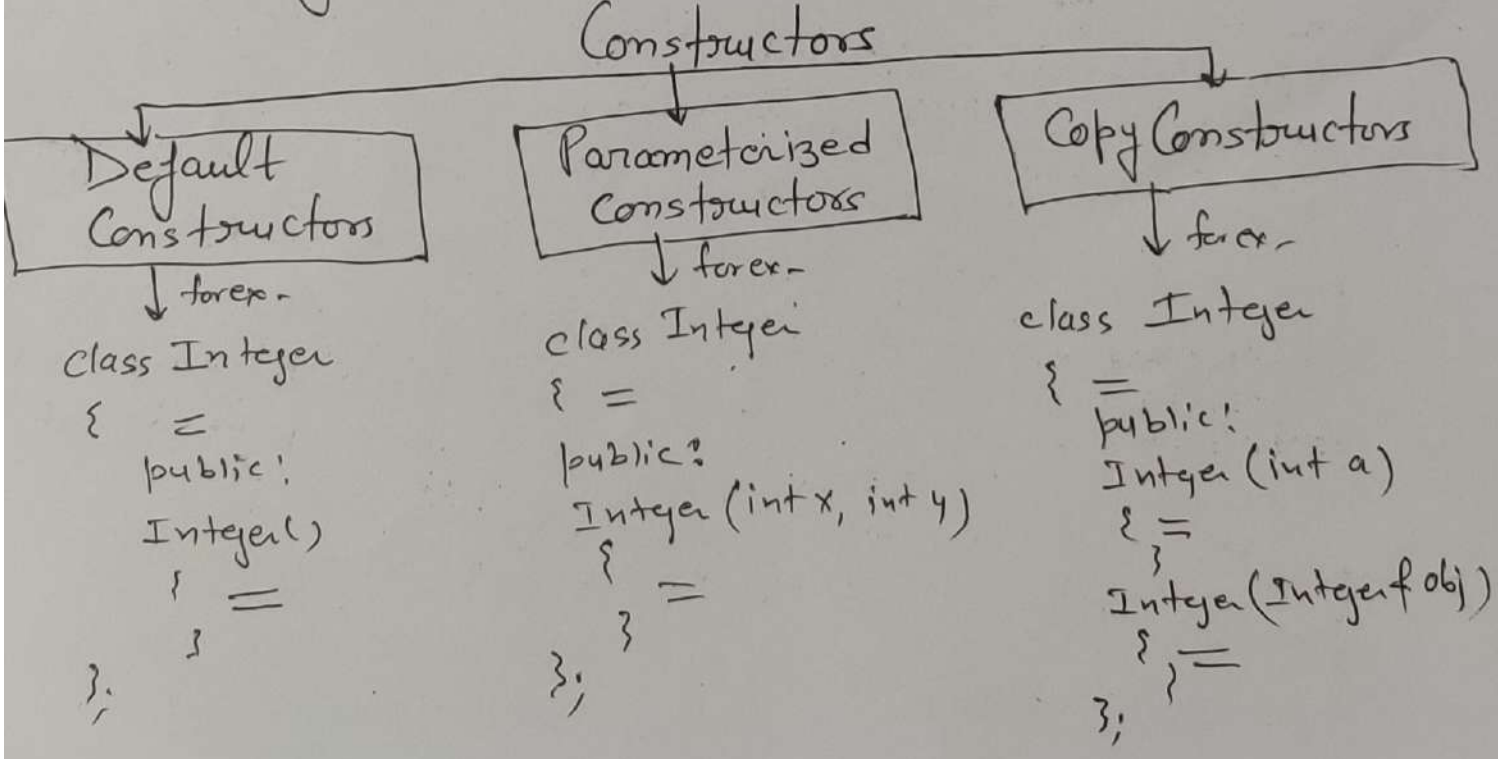
⇒ A constructor is declared and defined as —

```
// Class with a constructor
class integer
 {
    int a, b;
    public:
     integer();      // Constructor declared
     ....
     ....
 };
 integer :: integer()   // Constructor defined
   { a=0; b=0;
   }
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For ex-

```
integer obj;   // Object obj created & also
                  initialises its data members a & b.
```

Types of Constructors :-



① Default Constructor:-

⇒ A constructor that accepts no parameter is called the default constructor. the default constructor for class integer is —

integer :: integer()

⇒ If we define objects and classes without defining any constructor for a class in such situation, compiler automatically generate a default constructor.

integer a; // invoke default constructor of the compiler to create the object a.

Example:- Counter:- A counter is a variable that counts things, may be it counts file accesses, or the no. of times the user presses the (Enter) key, or the no of customer entering a bank. Each time such as an event takes place, the counter is incremented by 1 value. It can't also be accessed to find the current count

⇒ Let's assume that this counter is important in the program and must be accessed by many different functions. In procedural languages such as C, a counter would probably be implemented as an external variable. However, external variables complicate the program's design and may be modified accidentally.

⇒ This example, Counter, provides a counter variable that can be modified only through its member functions.

```
Class counter
{  int count;
  public:
   counter(): count(0)   //
    {  /* empty body */
    }
   void inc_count()   // increment count
    {  count++;
    }
   int get_count()   // return count
    {  return count;
    }
};
int main()
{
 Counter c1, c2;    // invoked Constructor
 cout << "\n c1 =" << c1.get_count();   // display
 cout << "\n c2 =" << c2.get_count();
 c1.inc_count();                  // increment c1
 c1.inc_count();
 c2.inc_count();                  // increment c2
 cout << "\n c1 =" << c1.get_count();  // display agin
 cout << "\n c2 =" << c2.get_count();
 cout << endl;
 return 0;
}
```

⇒ This Counter class has one data member: count and three member functions.—

the consfructer Counter()
inc_count() // add 1 to count value
get_count() // returns current value of count.

Parameterized Constructors :-

⇒ Default constructor initialises the data members of all objects to same value (i.e. 0. by compiler)

⇒ However, in practice it may be necessary to initialise the various data members of different objects with different values when they are created.

⇒ C++ allows us to achieve this objective by passing arguments to the constructor function when objects are created.

⇒ The Constructors that can take arguments are called parameterized constructors.

The Constructor integer() may be modified to take arguments as shown below —

```
Class integer
  {
    int a, b;
    public:
      integer (int x, int y);
      - - -
      - - -
  };
integer :: integer (int x, int y)
  { a = x; b = y; }
```

⇒ when constructor has been parameterised, object declaration statement such as -

```
integer obj;    // may not work
```

⇒ we must pass the initial values as arguments to the Constructor function when an object is declared.

```
integer obj1(7, 25);    // called constructor when obj1 is created
```

IInd way

```
integer obj1 = integer (7, 25);    // called Constru
```

Example:- A class "Point" that stores the x and y co-ordinate of a point. The class uses parameterized constructor for initializing the class objects.

```
Class Point
{  int x, int y;
   Public:
     Point (int a, int b);
     void Display ()
      { cout << "(" << x << "," << y << ")" << "\n";
      }
};
Point :: Point (int a, int b)
  { x = a; y = b; }

int main ()
{  Point P1 (2,2);
   Point P2 (5,7);
   P1. Display ();
   P2. Display ();
   return 0;
}
```

Copy Constructor:- A copy constructor is used to declare and initialize an object from another object.

for ex- statement integer $a_2(a_1)$; would define object a_2 and at the same time initialize it to the value of a_1. another form of this ~~obj~~ statement is — integer $a_2 = a_1$;

⇒ process of initializing through a copy Constructor is known as copy initialization

Example:- A copy constructor takes a reference to an object of the same class as itself as an argument.

⇒ Given class "Code" represents actual working of this constructor.

```
#include <iostream>
using namespace std;
Class Code
{  int id;
   public:
   Code()          // Constructor
    {
    }
   Code(int a)     // Constructor with one parameter
    { id = a; }
   Code(Code &i)   // Copy Constructor
    { id = i.id;   // Copy in values.
    }
};
void display(void)
  { cout<<id;
  }
int main()
{
  Code A(10); // A is created & initialized using Constructor
  Code B(A); // invoked Copy Constructor
  Code C = A; // again called copy constructor
  Code D;     // Object D is created
   D = A;     // Assigns value of A to D
cout<<"\n id of A:" ; A.display();
cout<<"\n id of B:" ; B.display();
cout<<"\n id of C:" ; C.display();
cout<<"\n id of D:" ; D.display();
  return 0;
}
```

⇒ We can not pass the argument by value to a copy constructor.

Destructors :-

⇒ Like a constructor, the destructor is a member function whose name is the same as the class name but it is preceded by a ~~tilde~~ tilde.

for ex- destructor for the class integer can be defined as—

```
~integer() {}
```

⇒ The most common use of destructors is to deallocate memory that was allocated for the objects by the constructor.

⇒ Destructor takes no arguments and has no return value.

implementation of Destructor :-

```
#include<iostream>
using namespace std;
int count =0;
class test
{ public:
    test()
    { count++;
      cout << "\n Constructor invoked : object no. " << count << "created";
    }
    ~test()
    {
      cout << "\n Destructor invoked : object no. " << count << "destroyed
      count--;
    }
};
int main ()
{ cout << " Inside the main block ...";
```

```
cout << "\n Creating first object T1|..";
    test T1;
    { // Block 1
      cout << "\n Inside Block 1..";
      cout << "\n Creating two more objects T2 and T3..";
      test T2, T3;
      cout << "\n Leaving Block 1..";
    }
  cout << "\n Back inside the main Block..");
   return 0;
}
```

⇒ Note that the objects are destroyed in the reverse order of their creation.

⇒ Finally, when the main block is exited, destructor is invoked corresponding to the remaining objects present inside main.

Polymorphism :-

⇒ Polymorphism is a combination of two Greek-words "Poly" and "morphs" which means many forms.

⇒ When same entity (function or object) behaves differently in different scenarios, it is known as "Polymorphism".

⇒ The best example of polymorphism is human behavior one person can have different behavior. for ex- a person acts as a customer in a shopping complex, a passenger in a train or bus, a student in a class or school and a son at a home.

⇒ It can be broadly classified into two categories as- shown in given figure :-

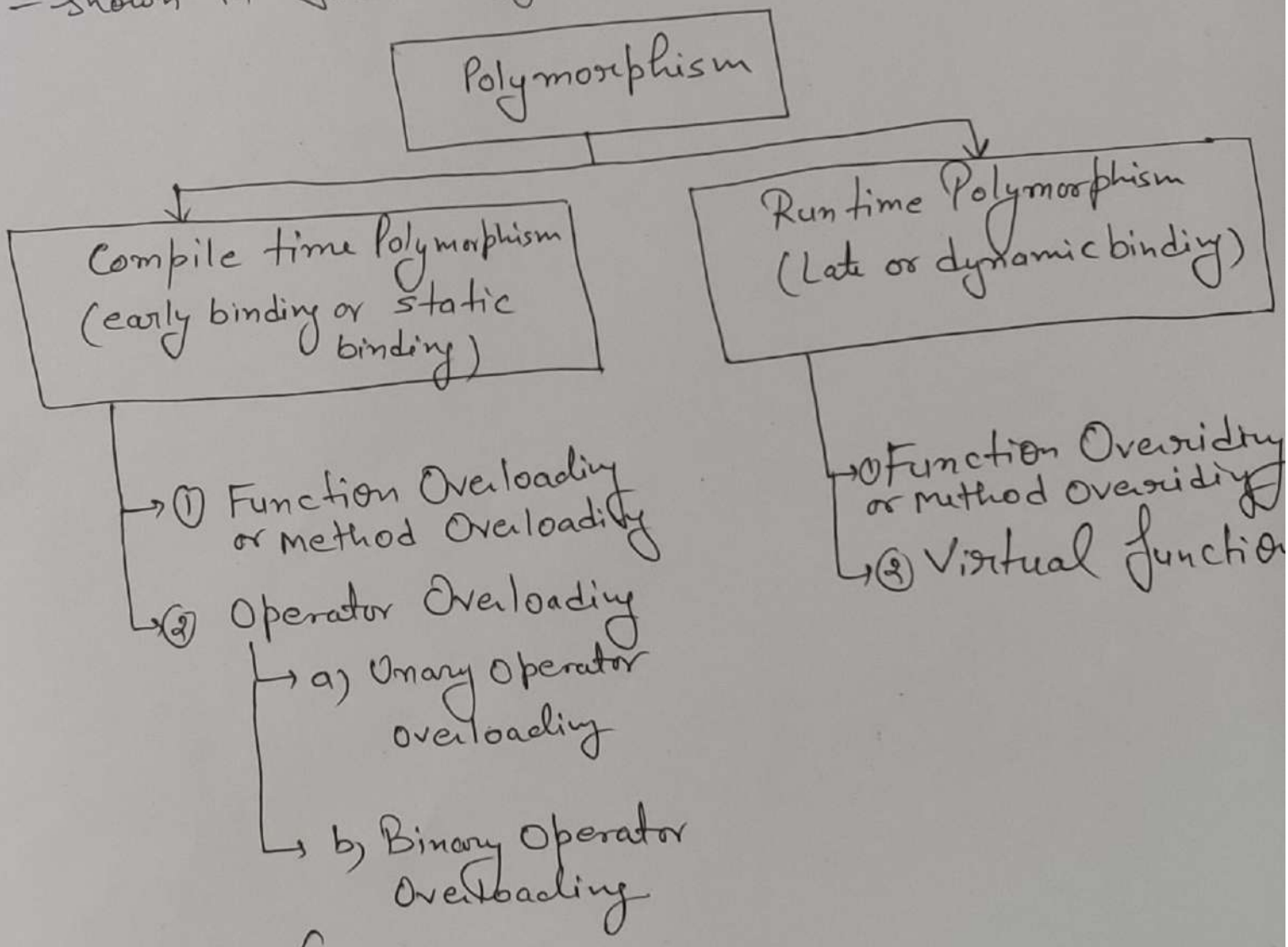


fig (Types of Polymorphism)

Compile-time Polymorphism :-

⇒ In Compile-time Polymorphism, a function is called at the time of program compilation. It means that an object is bound to its function call at the compile time.

⇒ There is no ambiguity at the compile time about which a function is to be linked to a particular function's call.

⇒ This mechanism is called early binding or static binding or static linking.

⇒ In C++, Compile-time polymorphism is achieved in two ways-

a) Function overloading
b) Operator overloading.

(a) Function Overloading :-

⇒ Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists.

⇒ The function would perform different operations depending on the argument list in the function call.

⇒ The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

⇒ The function selection involves the following steps:

(i) The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.

(ii) If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as -

char to int
float to double // to find the match.

(iii) When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compile will generate an error message -

⇒ Suppose we use the following two functions :-

```
long square (long n)
double Square (double x)
```

⇒ A function call such as - Square(10) // will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of Square() should be used.

(iv) If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match.

⇒ User-defined conversions are often used in handling class objects.

⇒ We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks.

(6) Operator Overloading :-

⇒ Overloading refers to the use of the same thing for different purposes. In operator Overloading, we can assign multiple meanings to operators.

⇒ for ex- The operator * when applied to a pointer variable, gives the value pointed to by the pointer. But it is also commonly used for multiplying two numbers.

⇒ The input/output operators << and >> are good examples of operator overloading. Although the built-in definition of the << operator is for shifting of bits, it is also used for displaying the values of various data types.

⇒ This has been made possible by the header file iostream where a number of overloading definitions for << are included. Thus the statement

```
cout << 75.86;   // invokes definition for displaying
                 a double type value,
```

and

```
cout << "well done";   // invokes definition for
                        displaying a char value.
```

However, none of these definitions in iostream affect the built-in meaning of the operator.

⇒ Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (. and .*), conditional operator (?:), scope resolution operator (::) and the size operator (sizeof).

Limitations in Operator Overloading :- There are certain restrictions & limitations in overloading them.

⇒ Sometimes, default arguments may be used instead of overloading. This may reduce the no. of functions to be defined.

⇒ The advantage of function overloading is that it increases the readability of the program because we don't need to use different names for the same action.

⇒ The overloaded function may or may not have a diffent data type but they must have different arguments.

// function area() is overloaded three times.

```
# include <iostream>
  int area(int);
  int area(int, int);
  float area(float);
{
 int main()
 { cout << "Area of a square :" << area(5) << endl;
   cout << "Area of a rectangle :" << area(5,10) << "\n";
   cout << "Area of a circle :" << area(5.5);
  return 0;
 }
int area(int a) // Area of a square
 { return(a * a);
 }
int area(int l, int b) // Area of rectangle
 {
  return (l * b);
 }
float area(float r) // Area of circle
 { return (3.14 * r * r);
 }
```

o/p

```
Area of a square: 25
Area of a rectangle: 50
Area of a Circle: 94.99
```

Overloading + Operator:- (functional notation into arithmetic notation)

```
#include<iostream>
using namespace std;
class complex
{
    float x;                    // real part
    float y;                    // imaginary part
    public:
    complex () { }              // 1st constructor
    Complex (float real, float imag)   // 2nd constructor
    {
        x = real;  y = imag;
    }
    Complex operator + (Complex);
    void display (void);
};
Complex complex :: operator + (complex c)
{   Complex temp ;              // temporary
    temp.x = x + c.x;           // float additions.
    temp.y = y + c.y;
    return (temp);
}
void complex :: display (void)
{
    cout << x << " + j" << y << "\n";
}
int main()
{ complex c1, c2, c3;           // invoke constructor 1
    c1 = complex (2.5, 3.5);    // invoke constructor 2
    c2 = complex (1.6, 2.7);
    c3 = c1 + c2;
  cout << "c1 =" ;  c1.display();
  cout << "c2 = " ;  c2.display();
  cout << "c3 = " ;  c3.display();
```

```
    return 0;
}
```

O/p of this program would be -

$C_1 = 2.5 + j3.5$

$C_2 = 1.6 + j2.7$

$C_3 = 4.1 + j6.2$

⇒ As a rule, in overloading of binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument.

⇒ Here, we can avoid the creation of temp object by replacing the entire function body by the following statement -

```
return complex ((x+c.x), (y+c.y));
```

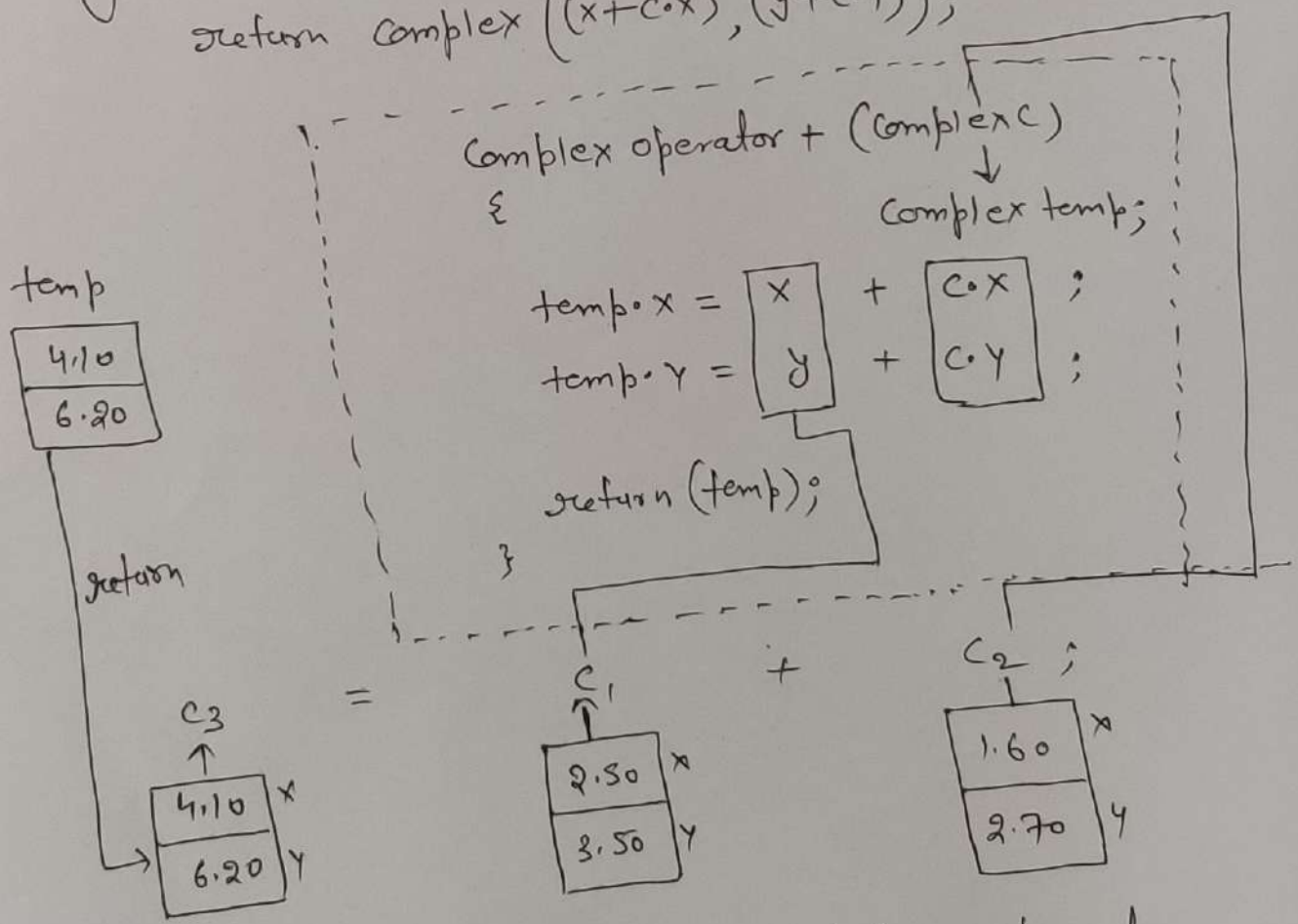


fig:- implementation of overloaded + Operator.