इंडियन बैंक Indian Bank

UNIT-2

Operator Overloading :-

a function can do multiple Oper

Follow Can't do on overloading in Operator

① class member access Operator (., .*) membership Oper → pointer-to-
② Scope resolution Operator (::)
③ Size operator (Sizeof)
④ Conditional Operator (?:).

Syntax

return type classname :: Operator op(arg list)
{
function body / task defined
}

1. unary
2. Binary

Features of this fun

1. It Receives only one complex type arg.
2. It returns of complex type value.
3. The member function of complex.

Rule :-

1. only existing Oper can be overloaded.
2. at least one operand must in user def type.
3. Can't used friend fun for certain operator.
4. It is not ~~overridding~~ overridden.

0547

CHMZ

Overloading Binary Operators using Friends

1. friend Complex operator+(complex, complex)
2. Complex operator + (complex a, complex b)

```
{
return complex((a.x+b.x),(a.y+b.y));
}
```

Type Conversion :-

Convert variable datatype from one datatype to another datatype

Same va

eg int m
float x = 3.14.
→ convert x is integer

data Conversion b/w un compatible type

1. conversion from basic type to class type
2. " " class type to basic type
3. " " One class type to another class

Basic to Class type

```
class time
{
    int hrs;
    int mins;
public:
    time(int t)
    { hours = t/60;
```

```
& main
{ time T1;
  int duration = 85;
  T1 = duration;
}
```

int to class type

ers to Basic type

Syntax

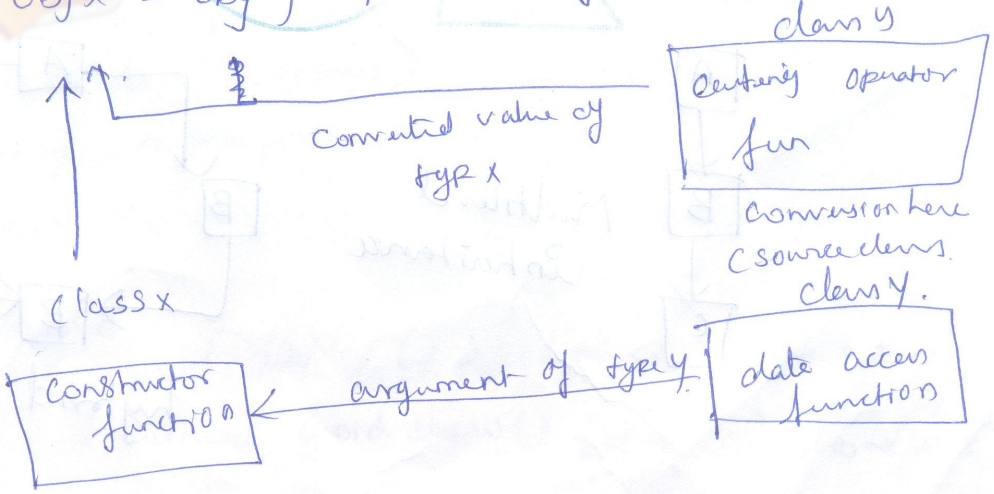
```
Operator typename ()
{
  ....
  ....  (fun stmt);
}
```

eg.

```
vector : Operator double ()
{
  double sum = 0;
  for (int i=0; i<size; i++)
    sum = sum + v[i] * v[i];
  return sqrt(sum);
}
```

One Class to Another Class type

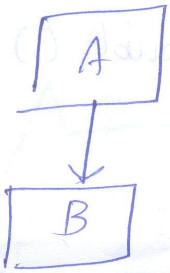
eg `objx = objy` // object of different type.



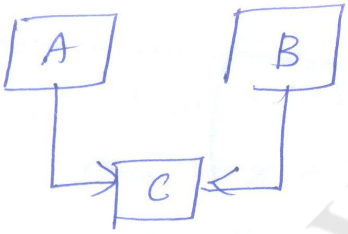
Inheritance.

deriving a new ~~cal~~ class from an old one is called inheritance.

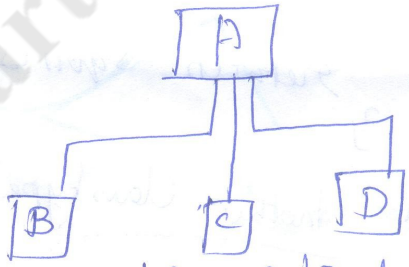
old class - base class.
new class - derived class.



Single inheritance.

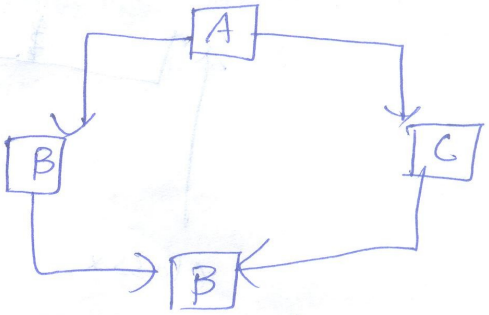


Multiple inheritance.



Hierarchical inheritance

Multilevel inheritance



hybrid inheritance.

class derived-class-name : visibility-mode base-class name.

{
... //
... // mem of derived class.
... //
};

Virtual function

polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms.

> Same fun name in both base and derived class.

> base class is declared as virtual.

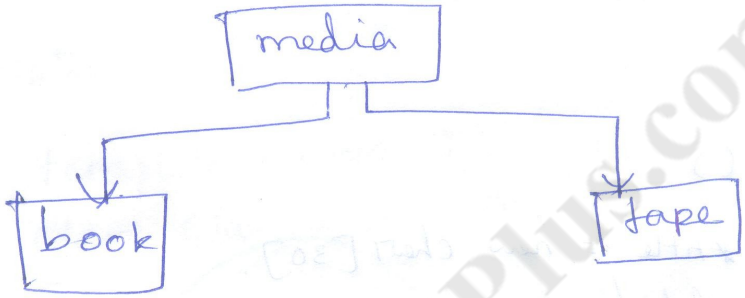
eg.

```
#include <iostream>
using namespace std;
class Base
{ public:
    void display () {cout << "Display base";}
    virtual void show() {cout << "Show base";}
};
```

```
class Derived : public Base.
{
  public:
      void display () { cout << " Display derived "; }
      void show() { cout << " show derived "; }
};
int main ()
{
    Base B;
    Derived D;
    Base *bptr;
  cout << " \n bptr points to base \n";
  bptr = &B;
  bptr -> display(); // call base version
  bptr -> show(); //
  cout << " bptr points to Derived \n";
  bptr = &D;
  bptr -> display(); // call base vers.
  bptr -> show(); // call derived version.
  return 0;
};
```

runtim polymorphism :-

It achieved only when a virtual function is accessed through a pointer to the base class.



eg pgm

```
# include <iostream>
# include <cstring>
using namespace std;
class media
{
  . . . .
public
    media ()
    {
      . . . .
    }
    virtual void display (){ }
};
class book : public media
 {
  public :                                  => create tape
    book (s,a,p) : media (s,a)
    {
```

```
void book :: display()
{
  cout << " title " << title        => Create tape
  "     "page"     "
}

int main ()
{ char *title = new char[30].
  // Book details.
  cout << " Title:" ; cin >> title;
  "       "price"       " price
  "       "pages"       " pages.
  book book1 (title, price, pages).
}
```

इंडियन बैंक
Indian Bank

Templates

A template can be used to create a family of classes or functions.

Class Templates

Syntax:

```
template <class T>
class classname
{
// class member specification
// with anonyms type T
// wherever appropriate
};
```

Class Templates with Multiple parameters

```
template <class T1, class T2, ...>
class classname
{
...(body of the class).
};
```

0547118
CHMZ

Function Templates

Syntax

```
template <class T>
returetype functionname (arg of typ
{
  // body of fun. with type T whener
  // appropriate
}.
```

Function Templates with multiple parameter

```
template <class T1, class T2 ... >
returtype functionname (arg of type T1, T2
{
  ... (body of function).
}
```

Overloading of Template function

1. Call an ordinary fun that has an exac mate
2. Call a template fun that could be create with an exact match.
3. Try normal overloading resolution to ord functions and call the one that matches.

(6)

इंडियन बैंक
Indian Bank

```
#include <iostream>
    <String>.

template <class T>
void display (T x)
{
 cout << " Tem disp : " << x << "\n";
}
void display (int x) // overload the generic display
{
  cout << " Explicit display : " << x << "\n";
}
int main ()
{
 display (100)
    " (12.34)
  d " ('c');
 return 0;
}
```

O/P
Explicit display : 100
Template " : 12.34.
" " : C

Member Function Templates

```
Template <class T>
returntype classname <T> :: functionname(ar
{
 //function body
}
```

(*) Friend functions & type conversions is missing

G. Devanathan
22/6/13
HOD/EEE.

→ This topic is included in hand written notes.
10/7/13

इंडियन बैंक
Indian Bank

Operator overloading:- → to provide an additional task to an existing operation. → It is convert one obj to another obj. → used as mem fun or friend function.

eg int a, b;
a+b;

time t1, t2
t1+t2 // adding obj.

Rules
→ existing operators can be overload.
→ It is user defined datatype.
→ basic meaning of operator should not be changed.
→ It follow syntax rules.
→ function call operator() → default arg.
→ Some operator can't simply overload.

Operators which cannot be overloaded:
> dot operator for member classes (.)
> The deref dereferences member to class operator (.*)
> Scope resolution operator (::)
> Size operator (sizeof).
> Conditional operator (?:)
> Casting operator (<>)
> # # tokens.

Operator can't overloaded as friends.
=, (), [], ->
↓ Assignment ↓ function call ↓ Array Subscript ↓ class member Access.

CHMZ 0547287

Mem fun syntax

```
return type operator operator_symbol (arg list)
{
 function body;
}
```

friend fun syntax.

```
friend return type operator operator_symbol (arg list)
{
 function body;
}.
```

Operator type	Operator function	
	Member fun	friend fun.
Uniary	no arg.	one reference object as a argument
Binary.	one arg	two reference object as argument.

Calling operator function.

operator type	Operator function.	
	Member fun	friend func
unary	op_sym obj; or obj op_sym;	obj = op_sym(obj2); (or) obj = (obj2) op_sym;
Binary.	obj1 oper_sym obj2	obj oper_sym obj2;

eg:-

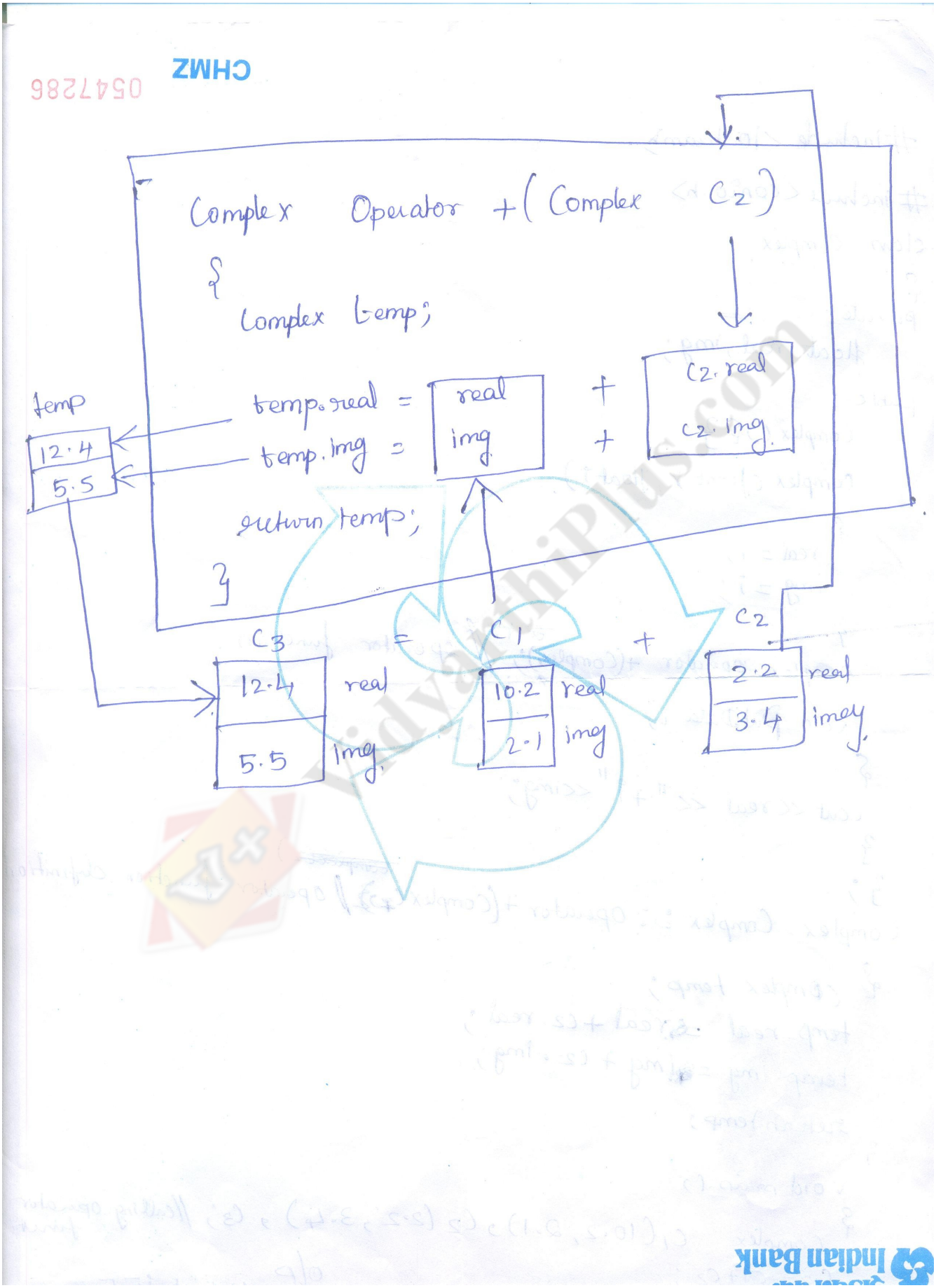
```
#include <iostream.h>
#include <conio.h>
class Complex
{
private:
    float real, img;
public:
    Complex(){}
    Complex(float r, float i)
    {
      real = r;
      img = i;
    }
    Complex operator +(Complex); // operator function.
    void PutData()
    {
    cout << real << "+i" << img;
    }
};
Complex Complex::Operator +(Complex c2) // operator function definit
{ Complex temp;
  temp.real = real + c2.real;
  temp.img = img + c2.img;
  return temp;
}
void main()
{
  Complex c1(10.2, 2.1), c2(2.2, 3.4), c3; //calling operat fun
```

o/p

eg:-

```
#include <iostream.h>
#include <conio.h>
class Complex
{
private:
    float real, img;
public:
    Complex(){}
    Complex(float r, float i)
    {
      real = r;
      img = i;
    }
    Complex operator +(Complex); // operator function.
    void PutData()
    {
    cout << real << "+i" << img;
    }
};
Complex Complex::Operator +(Complex c2) // operator function def
{ Complex temp;
  temp.real = real + c2.real;
  temp.img = img + c2.img;
  return temp;
}
  void main()
  {
    Complex c1(10.2, 2.1), c2(2.2, 3.4), c3; //calling op
    c3 = c1 + c2;
  } c3 = PutData();
```

o/p 12.4 + 15.5.



```
#include <iostream>
class sample
{ int x;
  public:
     Sample (int a)
     { x = a;
     }
     Operator int() // Conversion operator function.
     { return x;
     }
};
void main()
{ Sample S1(10);
  int a = S1; // Calling Conversion operator function.
  cout << a;
}
```

3) Conversion constructor.

```
class sample1
{ private:
     int x;
  public:
     Sample1(int a)
     { x = a;
     }
};
```

```
{ private: int y;
  public: Sample2() { }
     Sample2 (Sample1 S1) // Constructor used for Conversion.
     { y = S1.GetA(); } void print() { cout << y
};
void main() { Sample1 S1(10); Sample2 S2;
              S2 = S1; S2.print(); }
```

3) Conversion operator fun.

```
class one { int x; public: one() { }
  one (int a) { x = a; }
  void print() { cout << x; } };
class Two { int y; public: Two(int b)
  { y = b; }
  Operator one() { return one(y); }
};
void main()
{ one O1;
  Two T1(10);
  O1 = T1; // Calling the Conversion operator function
```

conversion operator fun — return to one any. constructor of one cl

Type Conversion.

It converting any given type to the type of current class.

Type conversion takes place b/w the incompatible data types.

1. Conversion from Built-in data type to an obj.
2. Conversion from obj to a Built-in datatype.
3. Conversion from one obj type to another obj type
   a) by a Conversion constructor
   b) by a Conversion operator function

Syntax:-

```
Operator type_name ()
{
function body;
}
```

1) It must be mem fun
2) No return type

1) Conversion from Built-in data type to object

```
#include <iostream>
    <conio.h>
class Sample.
{
  int x;
  public:
  Sample () { }
  Sample (int a) // Conversion Constructor.
  {
    x = a
  }
  void print () { cout << x; }
};
void main ()
{
Sample S1;
  S1 = 5
  S1.print(); // Conversion from int to object type.
}
```