OPERATOR OVERLOADING & TYPE CONVERSION

We can use operators like ‘+’ , ’-‘ , ’*’ , ’<’, etc. on built-in datatype like integer.

We can do same in User-defined datatypes, by Operator Overloading

DEFINATION: Giving special meaning to operator

IMPORTANT

- Only Semantics can be extended
- Other things will remain same Syntax, grammatical rules -> such as number of operands, precedence and associativity.

For example: ‘*’ will have higher precedence then ‘+’ operator

Hence, in OPERATOR OVERLOADING original meaning is not lost.

ALSO -> Operator Overloading must be used with User-Defined Datatype

HOW IS OPERATOR OVERLOADING DEFINED?

Either in Member function as

```
return type classname :: operator op(arglist)
{
      Function body          // task defined
}
```

where *return type* is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword **operator**. **operator** *op* is the function name.

E.g.:

student student :: operator+(student &x)

{…… Function body ……}

Or either as Friend Function.

MAIN TYPES OF OPERATOR OVERLOADING

- UNARY OPERATOR (Require only one operand, like not operator **!**)
- BINARY OPERATOR (Require 2 operand, like add **+**)

UNARY OPERATOR

Can be called using -> op x or x op

BINARY OPERATOR

Can be called using -> x op y

HOW WE OVERLOAD UNARY OPERATOR

MEMBER FUNCTIONS:

We need no arguments as this function can directly access data members of object calling this function.

Example on next page ->

```
class space
{
        int x;
        int y;
        int z;
 public:
        void getdata(int a, int b, int c);
        void display(void);
        void operator-();     // overload unary minus
};
void space :: getdata(int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void space :: display(void)
{
        cout << x << " ";
        cout << y << " " ;
        cout << z << "\n";
}
void space :: operator-()
{
        x = -x;
        y = -y;
        z = -z;
}

int main()
{
        space S;
        S.getdata(10, -20, 30);
        cout << "S : ";
        S.display();

        -S;                  // activates operator-() function

        cout << "S : ";
        S.display();

        return 0;
```

Remember, a statement like

```
S2 = -S1;
```

will not work because, the function **operator**–() does not return any value. It can work if the function is modified to return an object.

FOR FRIEND FUNCTION

It is possible to overload a unary minus operator using a friend function as follows:

```
friend   void operator-(space &s);              // declaration
void operator-(space &s)                        // definition
  {
        s.x = -s.x;
        s.y = -s.y;
        s.z = -s.z;
  }
```

note

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-(). Therefore, the changes made inside the operator function will not reflect in the called object.

HOW WE OVERLOAD BINARY OPERATOR

FOR MEMBER FUNCTION

```
#include <iostream>

using namespace std;

class complex
{
        float x;                              // real part
        float y;                              // imaginary part
  public:
        complex(){ }                          // constructor 1
        complex(float real, float imag)       // constructor 2
        { x = real; y = imag; }
        complex operator+(complex);
        void display(void);
};

complex complex :: operator+(complex c)
{
        complex temp;                 // temporary
        temp.x = x + c.x;             // these are
        temp.y = y + c.y;             // float additions
        return(temp);
}

void complex :: display(void)
{
        cout << x << " + j" << y << "\n";
}

int main()
{
        complex C1, C2, C3;           // invokes constructor 1
        C1 = complex(2.5, 3.5);       // invokes 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;
}
```

We can use

- **C3 = C1 + C2**
- **C3 = C1.operator+(C2)**

Hence LEFT HAND operand CALLS the function

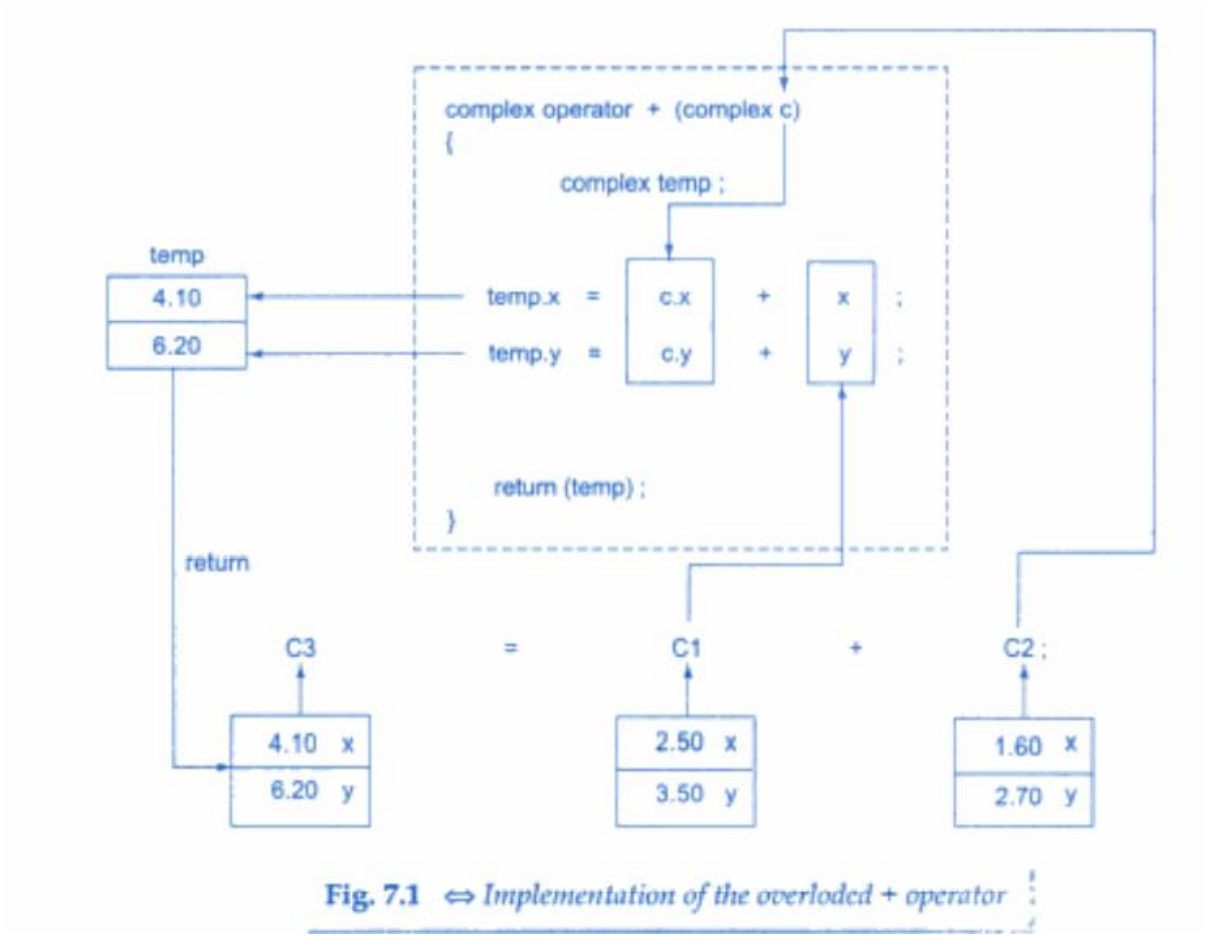

Fig. 7.1 ⇔ *Implementation of the overloaded + operator*

ALSO we can use this ->

```
return complex((x+c.x),(y+c.y));      // invokes constructor 2
```

Here we are creating a temporary object using parameterized constructor,

Using this makes code shorter, efficient, and more readable

USING FRIEND FUNCTION

Replace the member function declaration by the **friend** function declaration.

```
friend complex operator+(complex, complex);
```

Redefine the operator function as follows:

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

We can use

- **C3 = C1 + C2**
- **C3 = operator+(C1,C2)**

WHY USING FRIEND FUNCTION IN BINARY OPERATOR

See this case, if we had defined member function
A = B + 2; // Works fine
A = 2 + B; // Will not work as left operand is required to invoke the call, here it is int.

BUT friend function allows both approach

RULES FOR OPERATOR OVERLOADING (IMPORTANT)

- Only existing one can be overloaded; you cannot create new ones.
- At least one operand is required
- Cannot change original meaning, like '**+**' will ask subtraction
- **SOME OPERATORS CANNOT BE OVERLOADED**

Table 7.1 *Operators that cannot be overloaded*

Sizeof	Size of operator
.	Membership operator
.*	Pointer-to-member operator
::	Scope resolution operator
?:	Conditional operator

Table 7.2 *Where a friend cannot be used*

=	Assignment operator
( )	Function call operator
[ ]	Subscripting operator
->	Class member access operator

- In member function binary operator is invoked from left operand

TYPE CONVERSIONS

Basic type conversion like

float x = 3.14159;

int m = x;

Here x gets converted to int automatically (fractional part is truncated), BUT

how will this happen in User-Defined Datatype

CASES

- Basic type to class type
- Class type to basic type
- One class type to another class type

BASIC TO CLASS TYPE

➔ Easy to accomplish

➔ Constructor with one argument is used to accomplish this

Let us consider another example of converting an **int** type to a **class** type.

```
class time
{
    int hrs;
     int mins;
  public:
     ....
     ....
     time(int t)              // constructor
     {
          hours = t/60;       // t in minutes
          mins  = t%60;
     }
};
```

The following conversion statements can be used in a function:

```
time T1;                      // object T1 created
int duration = 85;
T1 = duration;                // int to class type
```

here we can see, **LEFT OPERAND before =** is responsible for invoking call and is always class object. We can use operator overloading unlike constructor in above example

CLASS TO BASIC TYPE

➔ We will use operator overloading the casting operator here… like double() or int()

Suppose we have a vector class

Consider the following conversion function:

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

How to invoke this.

```
double length = double(V1);
     or
double length = V1;
```

When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

RULES:

- Must be a member function of class
- It must not specify return type // User-defined conversions do not have a return type. C++ assumes you will be returning the correct type.
- It must not have any arguments // why? As it is already a member function

ONE CLASS TO ANOTHER CLASS TYPE

Example:

```
objX = objY;     // objects of different types
```

objX is an object of class **X** and **objY** is an object of class **Y**. The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**. Since the conversion takes place from **class Y** to **class X, Y** is known as the *source* class and **X** is known as the *destination* class.

➔ We can use constructor or conversion function
➔ Compiler treats them same way
➔ But when to use constructor and casting operator

CASTING OPERATOR -> operator typename()

Converts the object of which it is a typename (means destination class), here conversion takes place in source class and result is given to destination class!!

Constructor with single argument

It serves as an instruction for converting the argument's type to class type which it is a member.

Means argument belongs to source class, passed into destination class for conversion.

SUMMARIZATION OF TYPE CONVERSION

Table 7.3 *Type conversions*

Conversion required	*Conversion takes place in*	
	Source class	*Destination class*
Basic → class	Not applicable	Constructor
Class → basic	Casting operator	Not applicable
Class → class	Casting operator	Constructor

Will see example of type conversion in working code now!!