UNIT 3 OPERATOR OVERLOADING

3.0 INTRODUCTION

In the previous Unit, we discussed concept of a Class. In this Unit, we shall discuss the concept of overloading. For example, to add two matrices, you need to define a Matrix class. We will create two objects of this class. Now, when we have to add these two matrices, we have to access the data of their objects through member functions and add them. But, this is cumbersome. It will be convenient if I can add the two objects directly and the other details can be taken care of compiler. But, the '+' operator is meant for addition of data types like integers, etc. So, to add two objects with '+' operator, in C++, there is a facility to define a function with name '+' which accepts two objects as arguments. Then, we write the code in that procedure, which adds the data of these two objects and returns the object. This concept is known as operator overloading since single operator can be used for more than one purpose.

3.1 OBJECTIVES

After going through this Unit, you should be able to:

- write functions with operator overloading features;
- describe increment and decrement operators;
- describe concepts of subscripting and functions call.

3.2 OPERATOR FUNCTIONS

The meanings of the following operator can be redefined using functions:

+	-	*	/	%	^	&	|	~	!
=	>	<	+=	-=	*=	/=	%=	^=	&=
|=	<<	>>	>>=	<<=	= =	!=	<=	>=	&&
||	++	__	?*	,	?	[]	( )	New	Delete

Though the meanings are redefined, their precedence cannot be changed. At the same time, a Unary operator cannot be redefined as a Binary Operator.

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 =

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.

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 C
{
        C operator ++ (int);//Postfix increment
        C operator ++ ( ); //Prefix increment
        C operator || (C); //Binary OR
}
```

Some examples of Global Operator Functions are given below:

```
C operator – (C); // Prefix Unary minus
C operator – (C, C); // Binary "minus"
C operator – – (C&, 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.

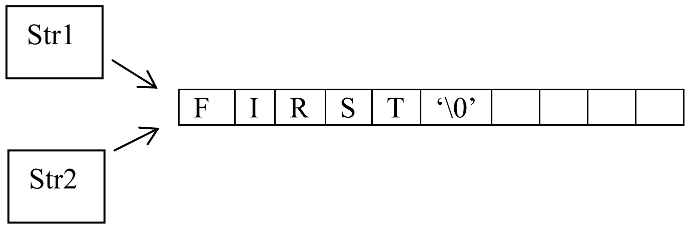
```
Class complex
{
        int real;
        int imag;
        public:
        friend complex operator + (const complex &x, const complex &y );
                        // operator overloading using friend
        complex ( ) { real = imag = 0;}
        complex (int x, int y) {real = x; imag = y;}
};
        complex operator + (const complex &x, const complex &y)
        {
                complex z;
                z.real = x.real +y.real;
                z.imag = x.imag + y.imag;
                return z;
}
        main ( ) {

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

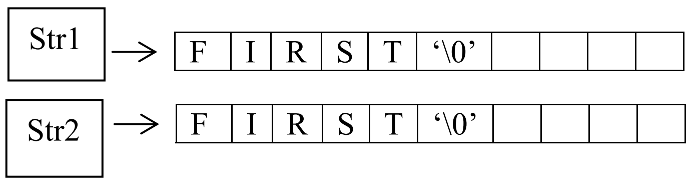
Operator overloading using member function:

```
        Class string
        {
        char * str;
        int length ; // Present length of the string
        int max_len; // (maximum space allocated to string)
        public:
        string ( );          // black string of length 0 of maximum allowed length
                             // of size 10.
        string (const string &s ) ;// copy constructor
        ~ string ( ) {delete str;}
        int operator = = (const string &s ) const; // check for equality
        string & operator = (const string &s );
        // overloaded assignment operator
        friend string operator + (const string &s1, const string &s2);
        } // string concatenation
        string::string ()
        {
        max_len = 10;
        str = new char [ max_len];
        length = 0;
        str [0] = '\0';
        }
string :: string (const string &s )
        {
        length = s. length;
        max_len = s.max_len;
        str = new char [max_len];
        strcpy (str, s.str); // physical copying in the new location.
        }
```

[**Comment:** 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.

```
String :: ~ string ( )
{
        delete str;
}
string & string :: operator = (const sting &s )
{
        if (this ! = &s) // if the left and right hand variables are different
        {
        length = s.length;
        max_len = s.max-len;
        delete str; // get rid of old memory space allocated to this string
        str = new   char [max_len];    // 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 string :: operator == (const sting &s ) const
        {
        // uses string comparison function
        return strcmp (str,s.str);
        }
string  string:: operator + (const string &s )
        string s3;
        s3.length = length + s.length;
        s3.max_len = s3.length;
        char * newstr = new char [len + 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 string &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.

3.3 LARGE OBJECTS

For all operator functions, which take classes as arguments, there is the overhead of copying entire object in the temporary storage. This can be avoided by declaring operator functions as taking reference arguments. For example, consider the following class matrix

```
class Matrix
{ int m [10] [10];
public:
Matrix ( );
        friend Matrix operator + (const Matrix&, const Matrix& );
        friend Matrix operator * (const Matrix&, const Matrix& );
};
```

Pointer cannot be used because it is not possible to redefine the meaning of an operator when applied to a pointer. The reference type, thus, avoids copying of large objects.

3.4 ASSIGNMENT AND INITIALISATION

Consider the following class:

```
class Employee
{
char name;
int ssno;
public:
Employee ( ) {name = new char [20];}
~Employee ( ) {delete [ ] name;}
};
int f ( )
{Employee E1, E2;
        cin >> E1;
        cin >> E2;
        E1 = E2;
}
```

Now, the problem is that after the execution of f ( ), destructors for E1& E2 will be executed. Since both E1 & E2 point to the same storage, execution of destructor twice will lead to error as the storage being pointed by E1 & E2 were disposed off during the execution of destructor for E1 itself.

Defining assignment of strings as follows can solve this problem,

```
class Employee
{
Public:
        char name;
        int ssno;
        Employee ( ) {name = new char [20];}
        ~Employee ( ) {delete [ ] name ;}
        Employee& operator = (const Employee& )
}
Employee & Employee :: Operator = (const Employee &e)
{
        if (this ! =&e)
```

```
            {
            delete [] name;
            name = new char [20];
            strcpy(name, name);
            }
            return *this;
}
```

3.5 SUBSCRIPTING

An operator [] function can be used to give subscripts, a meaning for class objects. The second argument (the subscript) of an operator [] function may be of any type.

Consider the following example which demonstrates the use of operator [] function:

```
#include <iostream.h>
Class item
{           int i;
    public:
            item ( )
                        {           cout<< "enter the number";
                                    cin>>i;
                         }
            int operator [ ] (int);
} ;

int item ::operator [ ] (int j )
{           if (i = = j)
            cout << "it matches";
            else
            cout << "it doesn't match";
            return 1 ;
}
void main ( )
{
            item array [8];
            for (int i=0; i<8; ++i)
            {           if (array [i][2])
                                cout << "operator overloading";
            }
}
```

3.6 FUNCTION CALL

Function call which is written as Procedure_name (argument1, argument2, ...) can also be interpreted as a binary operation with procedure_name as the left operand and arguments as the right operand. The call operator ( ) can be overloaded in the same way as other operators. An argument list for an operator ( ) function is evaluated and checked according to the usual argument passing rules.

The following example will demonstrate the operator ( ) function:

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

```
class data
{         char name [20];
          int index;
          public:
          data ( )
          {         cout << “Enter the name”;
                    cin >> name;
          }
          void operator ( ) (char);
};
void data :: operator ( ) (char source )
{         if (strcmp (source, name) == 0)
                    cout << “Matching occurred’’;
}

void main ( )
{
          data bank [10];
          char str [10];
          cout << “Enter the search string”;
                    cin >> str;
          for (int i=0; i<10; i++)
          {         bank [i] (str);
          }
}
```

3.7 INCREMENT

We can also overload “++” operator. Conventionally, since ++ can be used as postfix as well as prefix operator, we can have two different overloaded functions.

The following is an example of a program, which uses an operator function of ++ for prefix application:

```
#include <iostream.h>
class increment
{
          int i, j k ;
          public:
                    increment ( )
                    {
                              i=5;
                              j=6;
                              k=7;
                    }
void operator ++ ( )
{
          cout << (++i) << (++j) <<(++k);
}
};
void main ( )
{
increment in;
++in ;
}
```

The following is an example of an operator function of ++ for postfix application.

```
# include <iostream.h>
class increment
{
                int i, j, k;
        public:
                increment ( )
                 {
                        i = 5;
                        j=6;
                        k=7;
                }
void operator ++ (int )
{
        cout << (i++) << (j++) << (k++);
}
};
void main ( )
{
increment in;
in++;
}
```

Please note the prototype of prefix and postfix ++ operators.

3.8 DECREMENT OPERATOR

We can also overload -- operator. Conventionally, since -- can be used as a prefix as well as postfix operator, we can have two different overloaded functions.

The following is an example of a program, which uses an operator function of -- for prefix application.

```
#include <iostream.h>

class decrement
{
int i, j, k;
public:
        decrement ( )
        {
                i=5;
                j=6;
                k=7;
        }
void operator -- ( )
{
        cout << (--i) << (-- j) <<(--k);
}
};
void main ( )
{
decrement de;
--de;
}
```

The following is an example of a program, which uses an operator function of '– –' for postfix application.

```
#include <iostream.h>
class decrement
{
        int i, j, k;
        public:
                decrement ( );
        {
                i=5;
                j=6;
                k=7;
        }

void operator -- (int)
{
        cout << (i--) <<(j--)<<(k--);
)
);
void main ( )
{
decrement de;
-- de;
}
```

☞ **Check Your Progress**

1) All operators of C++ can be overloaded. True ☐ False ☐

2) There cannot be a member operator function, which receives other than _______as its left argument.

3) We can define our own storage allocations by overloading ______ and ___ ____.

4) The precedence of operators remains unchanged even if they are overloaded True ☐ False ☐

5) A Binary operator function may be defined for a unary operator function. True ☐ False ☐

3.9 SUMMARY

In this Unit, we have seen how to overload operators. All the operators that can be overloaded were listed in 3.2. Even after writing operator overloaded functions, the precedence of operators remains unchanged. Also, an operator that is unary cannot be used as a Binary operator by overloading. A '→' operator can be used as Postfix unary operator. The '++' & '--' operators can be used as Postfix or Prefix operators. So, separate functions overloading them for both the different applications have been shown. Finally, until the last Unit, we are of a view that Private data of a class can be accessed only in member functions of that class. But, other functions which are declared as "friend" can also access them.

3.10 SOLUTIONS/ ANSWERS

1) False
2) Object
3) New, Delete
4) True
5) False