UNIT 2   OPERATOR OVERLOADING

2.0   INTRODUCTION

When you create an object (a variable), you give a name to a region of storage.
A function is a name for an action. By making up names to describe the system at
hand, you create a program that is easier for people to understand and change.
Function overloading is one of the defining aspects of the C++ programming
language. Not only does it provide support for compile time polymorphism, it also
adds flexibility and convenience. In addition, function overloading means that if you
have two libraries that contain functions of the same name, they won't conflict as long
as the argument lists are different. We'll look at all these factors in detail across this
unit.

In C++, you can overload most operators so that they perform special operations
relative to classes that you create. When an operator is overloaded, none of its original
meanings are lost. Operator overloading allows the full integration of new class type
into programming environment. After overloading the appropriate operators, you can
use objects in expressions in just the same way that you use C++'s built-in data types.
Operator overloading also forms the basis of C++'s approach to I/O. Function
overloading and operator overloading really aren't very complicated. By the time you
reach the end of this unit, you shall learn when to use them and also underlying
mechanisms that implement them during compiling and linking.

2.1   OBJECTIVES

After going through this unit, you will be able to:

- explain the need of overloading;
- describe mechanism of function overloading;
- learn the use and application of default argument;

- understand how to overload (redefine) operators to work with new types;
- understand how to convert objects from one class to another;
- learn when to, and when not to, overload operators; and
- learn about several cases of overloaded operators.

2.2 FUNCTION OVERLOADING

Function overloading refers to using the same thing for different purposes. C++ permits the use of different functions with the same name. However such functions essentially have different argument list. The difference can be in terms of number or type of arguments or both. These functions can perform a variety of different tasks. This process of using two or more functions with the same name but differing in the signature is called function overloading. It is only through these differences that the compiler knows which function to call in any given situations.

Overloading of functions with different return types are not allowed.

Consider the following function:

```
int add (int a, int b)
{
return a + b;
}
```

This trivial function adds two integers. However, what if we also need to add two floating point numbers? This function is not at all suitable, as any floating point parameters would be converted to integers, causing the floating point arguments to lose their fractional values.

One way to work around this issue is to define multiple functions with slightly different names:

```
int add (int a, int b)
{
return a + b;
}
double addition (double p, double q)
{
return p + q;
}
```

However, for best effect, this requires that you define a consistent naming standard, remember the name of all the different flavors of the function, and call the correct one (calling add() for addition of double type numbers with integer parameters may produce the wrong result due to precision issues).

Function overloading provides a better solution. Using function overloading, we can declare another add () function that takes double parameters:

double add (double p, double q)

We now have two version of add ():

```
{
return p + q;
}
```

int add (int A, int B); // integer version

double add (double P, double Q); // floating point version

Which version of add () gets called depends on the arguments used in the call — if we provide two ints, C++ will know we mean to call add(int, int). If we provide two floating point numbers, C++ will know we mean to call add (double, double). In fact, we can define as many overloaded add () functions as we want, so long as each add () function has unique parameters.

Consequently, it's also possible to define add () functions with a differing number of parameters:

```
int add (int a, int b, int c)
{
return a + b + c;
}
```

Even though this add () function has 3 parameters instead of 2, because the parameters are different than any other version of add (), this is valid.

Function overloading is one of the most powerful features of C++ programming language. It forms the basis of polymorphism (compile-time polymorphism). Most of the time you'll be overloading the constructor function of a class.

2.2.1 How Function Overloading is Achieved

One thing that might be coming to your mind is, how will the compiler know when to call which function, if there are more than one function of the same name. The answer is, you have to declare functions in such a way that they differ either in terms of the number of parameters or in terms of the type of parameters they take. What that means is, nothing special needs to be done, you just need to declare two or more functions having the same name but either having different number of parameters or having parameters of different types. In overloaded functions, the function call determines which function definition will be executed. The biggest advantage of overloading is that it helps us to perform same operations on different datatypes without having the need to use separate names for each version. For example, an overloaded test() function handles different types of data as shown below:

```
// Declarations
int test (int x);                        //prototype 1
int test (int x, int y);                 //prototype 2
double test (int a, double b);           //prototype 3
// Function call
Cout<< test (23);                        //uses prototype 1
Cout<< test (45, 55);                    //uses prototype 2
Cout<< test (12, 3.25);        //uses prototype 3
```

Following example illustrates the function overloading:

```
// Example: Function overloading to find the absolute value of any number int,
long, float,
double
#include<iostream>
using namespace std;
int abslt(int );
long abslt(long );
float abslt(float );
double abslt(double );
int main()
{
int intgr=-5;
long lng=34225;
float flt=-5.56;
double dbl=-45.6768;
cout<<\" absoulte value of \"<<intgr<<\" = \"<<abslt(intgr)<<endl;
cout<<\" absoulte value of \"<<lng<<\" = \"<<abslt(lng)<<endl;
cout<<\" absoulte value of \"<<flt<<\" = \"<<abslt(flt)<<endl;
cout<<\" absoulte value of \"<<dbl<<\" = \"<<abslt(dbl)<<endl;
}
int abslt(int num)
{
if(num>=0)
return num;
else
 return (-num);
}
long abslt(long num)
{
if(num>=0)
return num;
else return (-num);
}
float abslt(float  num)
{
if(num>=0)
return num;
else return (-num);
}
double abslt(double  num)
{
if(num>=0)
return num;
else return (-num);
}
```

Output

- absoulte value of -5 = 5

- absoulte value of 34225 = 34225

- absoulte value of -5.56 = 5.56

- absoulte value of -45.6768 = 45.6768

The use of overloading may not have reduced the code complexity /size but has definitely made it easier to understand and avoided the necessity of remembering different names for each version function which perform identically the same task.

Example: overloading functions that differ in terms of number of parameters

```
#include<iostream.h>
// function prototype
int func(int i);
int func(int i, int j);
void main(void)
  {
  cout<<func(15);                 //func(int i)is called

  cout<<func(15,15);              //func(int i, int j) is called
  }
  int func(int i)
  {
  return i;
  }
  int func(int i, int j)
  {
  return i+j;
  }
```

Example: overloading functions that differ in terms of types of parameters

```
#include<iostream.h>
 //function prototypes
 int func(int i);
 double func(double i);
 void main(void)
 {
 cout<<func(15);                //func(int i) is called
 cout<<func(15, 155);           //func(double i) is called
 }
 int func(int i)
 {
 return i;
 }
double func(double i)
 {
 return i;
 }
```

39

2.2.2 How Function Calls are Matched with Overloaded Functions

Making a call to an overloaded function results in one of three possible outcomes:

1) A match is found. The call is resolved to a particular overloaded function.
2) No match is found. The arguments can not be matched to any overloaded function.
3) An ambiguous match is found. The arguments matched more than one overloaded function.

When an overloaded function is called, C++ goes through the following process to determine which version of the function will be called:

1) First, C++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions. For example:

```
void Print(char *szValue);
void Print(int nValue);
Print(10); // exact match with Print(int)
```

Although 10 could technically match Print(char*), it exactly matches Print(int). Thus Print(int) is the best match available.

2) If no exact match is found, C++ tries to find a match through promotion. In the lesson on type conversion and casting, we covered how certain types can be automatically promoted via internal type conversion to other types.

To summarize,

a) Char, unsigned char, and short is promoted to an int.
b) Unsigned short can be promoted to int or unsigned int, depending on the size of an int
c) Float is promoted to double
d) Enum is promoted to int

For example:

```
void Print(char *szValue);
void Print(int nValue);
Print('a'); // promoted to match Print(int)
```

In this case, because there is no Print(char), the char 'a' is promoted to an integer, which then matches Print(int).

3) If no promotion is found, C++ tries to find a match through standard conversion. Standard conversions include:

a) Any numeric type will match any other numeric type, including unsigned (eg. int to float)
b) Enum will match the formal type of a numeric type (eg. enum to float)
c) Zero will match a pointer type and numeric type (eg. 0 to char*, or 0 to float)
d) A pointer will match a void pointer

For example:

```
void Print(float fValue);
void Print(struct sValue);
Print('a'); // promoted to match Print(float)
```

In this case, because there is no Print(char), and no Print(int), the 'a' is converted to a float and matched with Print(float).

Note that all standard conversions are considered equal. No standard conversion is considered better than any of the others.

4) Finally, C++ tries to find a match through user-defined conversion. Although we have not covered classes yet, classes (which are similar to structs) can define conversions to other types that can be implicitly applied to objects of that class.

For example, we might define a class X and a user-defined conversion to int.

```
class X; // with user-defined conversion to int
void Print(float fValue);
void Print(int nValue);
X cValue; // declare a variable named cValue of type class X
Print(cValue); // cValue will be converted to an int and matched to Print(int)
```

Although cValue is of type class X, because this particular class has a user-defined conversion to int, the function call Print(cValue) will resolve to the Print(int) version of the function.

2.3   FUNCTION OVERLOADING AND AMBIGUITY

You can create a situation in which the compiler is unable to choose between two (or more) overloaded functions. When this happens, the situation is said to be ambiguous. Ambiguous statements are errors, and programs containing ambiguity will not compile.

2.3.1   Ambiguous Matches

If every overloaded function has to have unique parameters, how is it possible that a call could result in more than one match? Because all standard conversions are considered equal, and all user-defined conversions are considered equal, if a function call matches multiple candidates via standard conversion or user-defined conversion, an ambiguous match will result.

For example:

```
void Print(unsigned int nValue);
void Print(float fValue);
Print('p');
Print(10);
Print(1.14);
```

In the case of Print('p'), C++ can not find an exact match. It tries promoting 'a' to an int, but there is no Print (int) either. Using a standard conversion, it can convert 'a' to both an unsigned int and a floating point value. Because all standard conversions are considered equal, this is an ambiguous match.

Print(10) is similar. 10 is an int, and there is no Print(int). It matches both calls via standard conversion.

Print(1.14) might be a little surprising, as most programmers would assume it matches Print(float). But remember that all literal floating point values are doubles unless they have the 'f' suffix. 1.14 is a double, and there is no Print (double). Consequently, it matches both calls via standard conversion.

Ambiguous matches are considered a compile-time error. Consequently, an ambiguous match needs to be disambiguated before your program will compile. There are two ways to resolve ambiguous matches:

1) Often, the best way is simply to define a new overloaded function that takes parameters of exactly the type you are trying to call the function with. Then C++ will be able to find an exact match for the function call.

2) Alternatively, explicitly cast the ambiguous parameter(s) to the type of the function you want to call. For example, to have Print(10) call the Print(unsigned int),

2.4 MULTIPLE ARGUMENTS

If there are multiple arguments, C++ applies the matching rules to each argument in turn. The function chosen is the one for which each argument matches at least as well as all the other functions, with at least one argument matching better than all the other functions. In other words, the function chosen must provide a better match than all the other candidate functions for at least one parameter, and no worse for all of the other parameters.

In the case that such a function is found, it is clearly and unambiguously the best choice. If no such function can be found, the call will be considered ambiguous.

☞ **Check Your Progress 1**

1) What is function overloading?

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

2) How function calls are matched with overloaded functions?

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

3) What are the main applications of function overloading?

………………………………………………………………………………

………………………………………………………………………………

………………………………………………………………………………

4) Multiple choice questions:

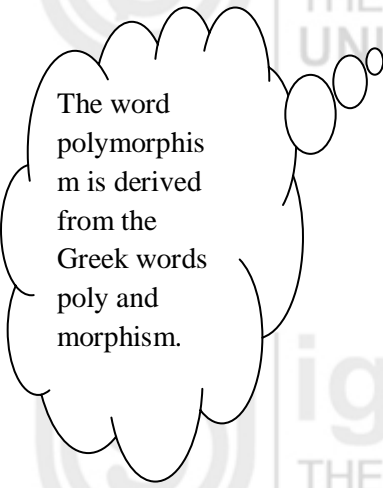   i)   In C++, dynamic memory allocation is accomplished with the operator ____

         a)   new

         b)   this

         c)   malloc( )

         d)   delete

   ii)   The operator that cannot be overloaded is

         a)   ++

         b)   ::

         c)   ( )

         d)   ~

   iii) The operator << when overloaded in a class

         a)   must be a member function

         b)   must be a non member function

         c)   can be both (A) & (B) above

         d)   cannot be overloaded

   iv)   Identify the operator that is NOT used with pointers

         a)   ->

         b)   &

         c)   *

         d)   >>

2.5   OPERATOR OVERLOADING

We already know that a function can be overloaded (same function name having multiple bodies). The concept of overloading a function can be applied to operators as well. For example, in C++ we can multiply two variables of user-defined data type with the same syntax that is applied to the basic data type. This means that C++ has the ability to provide the operators with a special meaning for data type. The

43

The word polymorphism is derived from the Greek words poly and morphism.

mechanism which provides this special meaning to operators is called operator overloading. The operator overloading feature of C++ is one of the methods of realizing polymorphism. Here, poly refers to many or multiple and morphism refers to actions, i.e. performing many actions with a single operator. Thus operator overloading enables us to make the standard operators, like +, -, * etc, to work with the objects of our own data types. So what we do is, write a function which redefines a particular operator so that it performs a specific operation when it is used with the object of a class. Operator overloading is very exciting feature of C++. The concept of operator overloading can also be applied to data conversion. It enhances the power of extensibility of C++. Thus operator overloading concepts are applied to the following two principle areas:

- Extending capability of operators to operate on user defined data, and
- Data conversion

This session deals with overloading of operators to make Abstract Data Types (ADTS) more natural, and closer to fundamental data types.

2.5.1 Why to Overload Operators

Most fundamental data types have pre-defined operators associated with them. For example, the C++ data type float, together with the operators +, -, *, and /, provides an implementation of the mathematical concepts of an integer. This is purely a convenience to the user of a class. Operator overloading isn't strictly necessary unless other classes or functions expect operators to be defined.

To make a user-defined data type as natural as a fundamental data type, the user-defined data type must be associated with the appropriate set of operators. Operators are defined as either member functions or friend functions. The purpose of operator overloading is to make programs clearer by using conventional meanings for ==, [], +, etc. Operators can be overloaded in any way from those available like globally or on the basis of class by class. While implementing the operator overloading this can be achieved by implementing them as functions. Whether it improves program readability or causes confusion depends on how well you use it. In any case, C++ programmers are expected to be able to use it.

The user can understand the operator notation more easily as compared to a function call because it is closer to the real-life implementation. Thus, by associating a set of meaningful operators, manipulation of an ADT can be done in a conventional and simpler form. Associating operators with an ADT involves overloading them. Although the semantics of any operator can be extend, we can't change its syntax, associativity, precedence etc.

2.5.2 Member vs. Non-member Operators

There are two groups of operators in terms of how they are implemented: member operators and non-member operators. The distinction between the two is the same as it is with methods and functions.

Member operators are operators that are implemented as member functions (methods) of a class.

Non-member operators are operators that are implemented as regular, non-member functions.

Some operators are required to be member operators, others must be non-member operators, and some can be both.

Note: Operator when overloaded is called operator function. Operator function is declared with operator keyword that precedes the operator that has to be overloaded. Again overloaded operator and functions are not the same, but the same way as overloaded functions are distinguished by the number and type of operands, the mechanism is applicable to the operators.

2.5.3 General rules for Operator Overloading

The following rules constrain how overloaded operators are implemented. However, they do not apply to the new and delete operators.

(i)   You cannot define new operators, such as **.

(ii)   You cannot redefine the meaning of operators when applied to built-in data types.

(ii)   Overloaded operators must either be a non static class member function or a global function.

(iii)   Obey the precedence, grouping, and number of operands dictated by their typical use with built-in types. Therefore, there is no way to express the concept "add 2 and 3 to an object of type Point," expecting 2 to be added to the x coordinate and 3 to be added to the y coordinate.

(iv)   Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.

(v)   Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.

(vi)   If an operator can be used as either a unary or a binary operator (&, *, +, and -), you can overload each use separately.

(viii) Overloaded operators cannot have default arguments.

(ix)   All overloaded operators except assignment (operator=) are inherited by derived classes.

(x)   The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

(xi)   Overloading + doesn't overload +=, and similarly for the other extended assignment operators.

(xii)   You may not redefine ::, sizeof, ?:, or . (dot).

(xiii) =, [], and -> must be member functions if they are overloaded.

(xiv)   ++ and -- need special treatment because they are prefix and postfix operators.

(xv)   There are special issues with overloading assignment (=). Assignment should always be overloaded if an object dynamically allocates memory.

Table 2.1 shows the operators which can be overloaded

Table 2.1: List of Operators that can be overloaded

+	-	*	/	%	^	&	|	~	!
=	<	>	+=	-=	*=	/=	%=	^=	&=
|=	<<	>>	<<=	>>=	==	!=	<=	>=	&&
||	++	--	,	->*	->	( )	[ ]	new	delete
new[]	delete[]								

Following list shows the operators which can't be overloaded

.	.*	::	sizeof	?:

2.5.4 Why Some Operators can't be Overload

Generally the operators that can't be overloaded are like that because overloading them could and probably would cause serious program errors or it is syntactically not possible. Following section describe the reason for not overloading some of the operators defined in C++ language.

a) :: (scope resolution)), . (member selection), and .* (member selection through pointer to function)

For the operators that cannot be overloaded like :: (scope resolution), . (member selection), and .* (member selection through pointer to function), we are quoting from Stroustrup's 3rd edition of 'The C++ Programming Language', section 11.2 (page 263), these operators 'take a name, rather than a value, as their second operand and provide the primary means of referring to members. C++ has no syntax for writing code that works on names rather than values so syntactically these operators can not be overload.

The right hand side of operators **.** (member selection/access) , .* (member selection through pointer to function) and :: (scope resolution) are names of things, e.g. name of a class member. In other words: above three unloaded operators use name instead of operand, so we can`t pass any name (either of variable, class) to any function. We must have to pass the operand for that.

b) **size of operator**

The size of operator returns the size of the object or type passed as an operand. It is evaluated by the compiler not at runtime so you can not overload it with your own runtime code. It is syntactically not possible to do.

c) **? : (conditional operator)**

All operators that can be overloaded must have at least one argument that is a user-defined type. That means you can't overload that operator which has no arguments.

But it does not suite for ? : (conditional operator) as it does not take name as parameter. The reason we cannot overload ? : is that it takes 3 argument rather than 2 or 1. There is no mechanism available by which we can pass 3 parameters during operator overloading.

2.6 DEFINING OPERATOR OVERLOADING

You can overload or redefine the most of built-in operators in C++. These operators can be overloaded globally or on a class-by-class basis. Overloaded operators are implemented as functions and can be member functions or global functions. An overloaded operator is called an operator function. You declare an operator function with the keyword operator preceding the operator. Overloaded operators are distinct from overloaded functions, but like overloaded functions, they are distinguished by the number and types of operands used with the operator.

2.6.1 Syntax

Defining an overloaded operator is like defining a function, but the name of that function is operator #, in which # represents the operator that's being overloaded. The number of arguments in the overloaded operator's argument list depends on two factors:

1. Whether it's a unary operator (one argument) or a binary operator (two arguments).
2. Whether the operator is defined as a global function (one argument for unary, two for binary) or a member function (zero arguments for unary, one for binary – the object becomes the left-hand argument).

It may look like following way:

```
Return_type  class_name :: operator op (op_argument_list)
{
Body of function
}
```

Here 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 name of function. They may be also friend functions. Member function has no argument for unary operator and one argument for binary operator. This is because the object used to invoke the member function is passed implicitly and so it available to member function. This case is not with the friend function. Friend function will have one argument for unary operator and two arguments for binary operator. All the arguments may be passed either by value or by reference. Following lines define the steps in overloading the operators

a. Build a class that defines the data type that is going to use in operation of overloading.
b. Declare the operator function *operator op()* in public area of class.
c. Now define the operator function to implement the required operations.

Invoking Operator Function

(i) For member Function
      a. For unary operator: **op m** or **m op**
      b. For binary operator: **m op n** or **m.operator op (n)**

(ii) For friend Function
    a. For unary operator: **operator op (m)**
    b. For binary operator: **operator op (m, n)**

Overloading unary operators

To declare a unary operator function as a non static member, you must declare it in the
form:
return_type **operator** *op*()**;**
where *op* is one of the operators listed in the preceding table.
To declare a unary operator function as a global function, you must declare it in the
form:
return_type **operator** *op*(*arg*)**;**
Where *op* is described for member operator functions and the *arg* is an argument of
class type on which to operate. An overloaded unary operator may return any type.

Some examples of operator overloading:

Unary operator overloading
Example:

```
#include<iostream.h>
#include<conio.h>
Class complex
{
int real, imaginary;
Public:
complex()
{
}
complex(int a, int b)
{
real = a;
imaginary = b;
}
void operator-();
void display()
{
cout<<"Real value"<<real<<endl;
cout<<"imaginary value"<<imaginary<<endl;
}
};
void complex::operator-()
{
real = -real;
imaginary = -imaginary;
}
void main()
{
Clrscr();
complex c1(10,12);
cout<< "real and imaginary value befor operation"<<endl;
c1.display();
c1;       //c1- /*It will give error*/
cout<< "real and imaginary' value after operation"<<endl;
c1.display();
getch();
}
```

Output

Real and imaginary value before operation
10   20
Real and imaginary value after operation
-10   -20

Explanation: In the above example operator, overloading is of unary operator declared in class complex. Outline function make changes values of real and imaginary part by negation. When this is called, it has to be remembered that operand must precede operator otherwise there will be an error. The reason is simple, 'operator' is the function name that is called with operand.

Unary operator overloading using friend function

Example:

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real;
int imaginary;
public:
  complex()                        //default constructor
{
}
complex(int a, int b)
{
        real  =  a;
        imaginary  =  b;
}
friend void operator -(); //operator overloading prototype
void display()
 {
cout<<"real value is"<<real<<endl;
cout <<"imaginary value is:"<<imaginary<<endl;
 }
};
//definition of operator overloading function
friend void complex :: operator - (complex &c)
{
        c.real  =  - c.real;
        c.imaginary  =  - c.imaginary;
}
void main()
{
clrscr();
complex c1(50,100);
cout<<"real and imaginary value before operation"<<endl;
c1.diaplay();      //calling operator overloading function
-c1;
//c1-;              It will give you an error.
cout<<"real and imaginary value after operator"<<endl;
 c1.display();
 getch();
}
```

49

Output:

Real and imaginary value before operation

real value is 50   and imaginary value 100

real and imaginary value after operation

real value is -50   and imaginary value -100

2.7.2 Binary Operator Overloading

Example :

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real, imaginary;
public:
complex()
{
}
complex(int a,int b)
{
real  =  a;
imaginary  =  b;
}
void operator +(complex c);
};
void complex::operator+(complex c)
{
complex temp;
temp.real = real + c.real;
temp.imaginary = imaginary + c.imaginary;
cout<<"real sum is"<<temp.real<<endl;

}
void main()
{
clrscr();
complex c1(10,20);
complex c2(20,30);
c1+c2;
getch();
}
```

Output

Real sum is   30

Imaginary sum is 50

Explanation: Output is easily predictable, and most of the program is same as above program. Difference lies in void complex :: operator +(complex c) if we compare this statement with our conventional outline statement

<return type > <classname>::<function name> <argument list>

We find that return type is void. Class name is complex. Function name is 'operator+' and it takes one argument which is of class complex type object.

When this function is called in main with statement c1+c2, c1 is object that invokes function 'Operator+' and c2 is passed as argument. Now with in the function real and imaginary parts are c1 will be directly accessible (as it invokes function) while real and imaginary of c2 are taken using formal argument 'complex c'. Here we can find that keyword 'operator' is inserted automatically whenever invoking object is user defined type with operator.

Binary operator overloading using friend function

Example

```
#include<iostream.h>
#include<conio.h>
class complex
{
int real;
int imaginary;
public:
  complex(){ }                    //default constructor
complex(int a, int b)
{
        real = a;
        imaginary = b;
}
friend complex operator +(comlex c1, complex c2);
void display()
 {
cout<<"real value is"<<real<<endl;
cout <<"imaginary value is:"<<imaginary<<endl;
 }
};
complex  operator + (comlex c1, complex c2)
{
        complex tmp;
        tmp.real =  c1.real + c2.real
        tmp.imaginary =  c1.imaginary + c2.imaginary;
        return(tmp);
}
void main()
{
clrscr();
complex c1(10,20);
complex c1(30,50);
ccomlex c3 = c1 + c2;
c3.display();
 getch();
 }
```

Output:

real value is : 40

imaginary value is : 70

Unary Operators (Increment and decrement)

The overloaded ++ and − − operators present a dilemma because you want to be able to call different functions depending on whether they appear before (prefix) or after (postfix) the object they're acting upon. The solution is simple, but people sometimes find it a bit complex and confusing at first. When the compiler sees, for example, ++a (a pre-increment), it generates a call to operator ++(a); but when it sees a++, it generates a call to operator ++(a, int.) That is, the compiler differentiates between the two forms by making calls to different overloaded functions.

The following example overloads the unary operators:

Example

```cpp
#include<iostream>
using namespace std;
//Increment and decrement overloading
class Inc {
        private:
                int count ;
        public:
                Inc() {
                        //Default constructor
                        count = 0 ;       }
                Inc(int C) {
                        // Constructor with Argument
                        count = C ;
                }
                Inc operator ++ () {
                        // Operator Function Definition
                        return Inc(++count);
                }
                Inc operator -- () {
                        // Operator Function Definition
                        return Inc(--count);
                }
                void display(void) {
                        cout << count << endl ;
                }
};
```

```
void  main(void)
 {
        Inc a, b(10), c, d, e(5), f(10);
        cout << "Before using the operator ++()\n";
        cout << "a = ";
        a.display();
        cout << "b = ";
        b.display();
        ++a;
        b++;
        cout << "After using the operator ++()\n";
        cout << "a = ";
        a.display();
        cout << "b = ";
        b.display();
        c = ++a;
        d = b++;
        cout << "Result prefix (on a) and postfix (on b)\n";
        cout << "c = ";
        c.display();
        cout << "d = ";
        d.display();
        cout << "Before using the operator --()\n";
        cout << "e = ";
        e.display();
        cout << "f = ";
        f.display();
        --e;
        f--;
        cout << "After using the operator --()\n";
        cout << "e = ";
        e.display();
        cout << "f = ";
        f.display();
 }
```

Output:

Before using the operator ++()

a = 0

b = 10

After using the operator ++()

a = 1

b = 11

Result prefix (on a) and postfix (on b)

c = 2

53

d = 12

Before using the operator --()

e = 5

f = 10

After using the operator - -()

e = 4

f = 9

Note :

When specifying an overloaded operator for the postfix form of the increment or decrement operator, the additional argument must be of type int; specifying any other type generates an error.

Overloading assignment operator ('=')

```cpp
class Complex
{
private:
        double real, imag;
public:
        Complex(){
                real = 0;
                imag = 0;
        }
        Complex(double r, double i) {
                real = r;
                imag = i;
        }
        double getReal() const {
                return real;
        }
        double getImag() const {
                return imag;
        }
        Complex & operator=(const Complex &);
};
Complex & Complex::operator=(const Complex& c) {
        real = c.real;
        imag = c.imag;
        return *this;
}
#include <iostream>
int main()
{
        using namespace std;

        Complex c1(5,10);
Complex c2(50,100);
        cout << "c1= " << c1.getReal() << "+" << c1.getImag() << "i" << endl;
```

```
        cout << "c2= " << c2.getReal() << "+" << c2.getImag() << "i" << endl;
        c2 = c1;
        cout << "assign c1 to c2:" << endl;
        cout << "c2= " << c2.getReal() << "+" << c2.getImag() << "i" << endl;
}
```

Output:

c1= 5+10i
c2= 50+100i
assign c1 to c2:
c2= 5+10i

Overloading the << Operator

Output streams use the << operator for standard types. We can also overload the << operator for our own classes.

Actually, the << is left shift bit manipulation operator. But the ostream class overloads the operator, converting it into an output tool. The cout is an ostream object and that it is smart enough to recognize all the basic C++ types. That's because the ostream class declaration includes an overloaded operator<<() definition for each of the basic types.

Example

```
#include <iostream>
using namespace std;
class Complex
{
private:
        double real, imag;
public:
        Complex(){
                real = 0;
imag = 0;
        }
        Complex(double r, double i) {
                real = r;
                imag = i;
        }
        double getReal() const {
                return real;
        }
        double getImag() const {
                return imag;
        }
        Complex & operator=(const Complex &);
        const Complex operator+(const Complex & );
        Complex & operator++(void);
        Complex  operator++(int);
        /*friend const
                Complex operator+(const Complex&, const Complex&); */
        friend ostream& operator<<(ostream& os, const Complex& c);
};
```

```cpp
Complex & Complex::operator=(const Complex& c) {
        real = c.real;
        imag = c.imag;
        return *this;
}
const Complex Complex::operator+(const Complex& c) {

Complex temp;
        temp.real = this->real + c.real;
        temp.imag = this->imag + c.imag;
        return temp;
}
//pre-increment
Complex & Complex::operator++() {
        real++;
        imag++;
        return *this;
}
//post-increment
Complex Complex::operator++(int) {
            Complex temp = *this;
            real++;
            imag++;
            return temp;
}
/* This is not a member function of Complex class */
/*const Complex operator+(const Complex& c1, const Complex& c2) {
            Complex temp;
            temp.real = c1.real + c2.real;
            temp.imag = c1.imag + c2.imag;
            return temp;
}*/
ostream& operator<<(ostream &os, const Complex& c) {
        os << c.real << '+' << c.imag << 'i' << endl;
        return os;
}
int main()
{
        Complex c1(5,10);
        cout << "c1 = " << c1.getReal() << "+" << c1.getImag() << "i" << endl;
        cout << "Using overloaded << " << endl;
        cout << "c1 = " << c1 << endl;
}
```

Output:

c1 = 5+10i

Using overloaded <<

c1 = 5+10i

Note that we just used:

cout << "c1 = " << c1 << endl;

Note that when we do

cout << c1;

it becomes the following function call:

operator<<(cout, c1);

operator overloading for strings

```
#include<iostream.h>
 #include<string.h>
 #include<conio.h>
class string
   {
     private:
      char str[80];
     public:
      string()  { strcpy(str,"ttt"); }
string(char s[])  { strcpy(str,s); }
      void display()  { cout<<str<<endl; }
         string operator+(string );
   };
string string::operator+(string ss){
     string temp;
     if(strlen(str)+strlen(ss.str)<80)
     {
      strcpy(temp.str,str);
      strcat(temp.str,ss.str);
     }
     else
     {
      cout<<"string overflow"<<endl;
      temp=0;
     }
     return temp;     }
 main()
   {
     clrscr();
     string s1(" Operator");
     string s2(" Overloading");
     string s3;
     s1.display();
     s2.display();
     s3=s1+s2;
     s3.display();
     getch();
     return 0;
   }
```

Output:

Operator

Overloading

Operator Overloading

Operator overloading from String object to basic string

```cpp
#include <iostream.h>
#include <string.h>
#include <conio.h>
class string
{
  char *p;
  int len;
  public:
   string()
   {}
   string(char *a)
   {
    len=strlen(a);
    p=new char[len+1];
    strcpy(p,a);
   }
   operatorchar*()
   {
    return(p);
   }
   void display()
   {
    cout<<p;
   }
};
void main()
{
 clrscr();
 string o1="IGNOU";
 cout<<"String of Class type : ";
 o1.display();
 cout<<endl;

 char *str=o1;
 cout<<"String of Basic type : "<<str;
 getch();
}
```

Output:

String of Class type : IGNOU

String of Blass type : IGNOU

Instream and outstream operator overloading

Example

```
Class Complex
{
Public:
Friend istream & operator >>(istream &is, Complex &c2);
Friend ostream & operator <<(ostream &os, Complex &c2);
Private:
Int real,imaginary;
};
Istream& operator >>(istream &is, Complex &c2);
{
Cout<<"enter real and imaginary"<<endl;
Is>>c2.real>>c2.imaginary;
Return(is);
}
Istream& operator <<(ostream &os, Complex &c2)

{
Os<<"the complex number is "<<endl;
Os<<c2.real<<"I"<<c2.imaginary;
Return(os);
}
Void main()
{
Complex c1,c2;
Cin>>c1;
Cout<<c1;
Cin>>c2;
Cout<<c2;
}
```

This operator function have to be declared friends since they have to access the user class and the objects of stream and ostream classes that are sytem defined. Since these operators functions are friend functions, the two objects cin and cout are passed as arguments, along with the objects of the user class. They return the isteam and ostream objects so that the operator can be chained. That is the above two input statements can also be written as,

Cin>>c1>>c2;

Cout<<c1<<c2;

☞ **Check Your Progress 2**

1) What is concept of operator overloading?

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

2) What are the basic rules for operator overloading in C++?

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

3) What are the limitations of Operator overloading and Functional overloading?

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

4) Multiple choice questions:

i)   Which of the following statements is NOT valid about operator overloading?

a)   Only existing operators can be overloaded.

b)   Overloaded operator must have at least one operand of its class type.

c)   The overloaded operators follow the syntax rules of the original operator.

d)   none of the above.

ii)   The new operator

a)   returns a pointer to the variable

b)   creates a variable called new

c)   obtains memory for a new variable

d)   tells how much memory is available

iii)   Which of the following operator can be overloaded through friend function?

a)   ->

b)   =

c)   ( )

d)   *

iv)   Overloading a postfix increment operator by means of a member function takes

a)   no argument

b)   one argument

c)   two arguments

d)   three arguments

2.7   SUMMARY

In this unit, we have discussed two important concepts about overloading namely function overloading and operator overloading. As a part of function overloading, you learnt that you can create multiple functions of the same name that work differently depending on parameter type. Function overloading can lower a programs complexity significantly while introducing very little additional risk. Although this particular lesson is long and may seem somewhat complex (particularly the matching rules), in reality function overloading typically works transparently and without any issues. The compiler will flag all ambiguous cases, and they can generally be easily resolved. Operator overloading is common-place among many efficient C++ programmers. Operating overloading allows you to pass different variable types to the same function and produce different results. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type. Operator overloading allows the programmer to define how operators should interact with various data types. It is so because operators in C++ are implemented as functions, operator overloading works very analogously to function overloading. Operator overloading is the ability to tell the compiler how to perform a certain operation when its corresponding operator is used on one or more variables.

2.8   ANSWERS TO CHECK YOUR PROGRESS

Check Your Progress 1

1)   C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as their types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

2)   a) A match is found. The call is resolved to a particular overloaded function.
b) No match is found. The arguments can not be matched to any overloaded function.
c) An ambiguous match is found. The arguments matched more than one overloaded function.

When an overloaded function is called, C++ goes through the following process to determine which version of the function will be called:

i)   First, C++ tries to find an exact match. This is the case where the actual argument exactly matches the parameter type of one of the overloaded functions. For example:

void Print(char *szValue);

void Print(int nValue);

Print(10); // exact match with Print(int)

Although 10 could technically match Print(char*), it exactly matches
Print(int). Thus Print(int) is the best match available.

ii) If no exact match is found, C++ tries to find a match through promotion.
In the lesson on type conversion and casting, we covered how certain
types can be automatically promoted via internal type conversion to other
types.

3) a) The use of function overloading is to increase consistency and readability.

b) Overloading provides multiple behaviour to same object with respect to
attributes of object.

c) By using function overloading, we can call specific behaviour of that
object attributes we set at compiled time. Further, we don't need to write
different function name for different action.

d) can develop more than one function with the same name.

e) function overloading exhibits the behavior of polymorphism which helps
to get different behaviour, although there will be some link using same
name of function. Another powerful use is constructor overloading, which
helps to create objects differently and it also helps a lot in inheritance.

4) Multiple Choice Questions

i) (A)

ii) (B)

iii) (C)

iv) (D)

Check Your Progress 2

1) Operator overloading allows existing C++ operators to be redefined so that
they work on objects of user-defined classes. They form a pleasant facade that
doesn't add anything fundamental to the language (but they can improve
understandability and reduce maintenance costs). When an operator is
overloaded, it takes on an additional meaning relative to a certain class. But it
can still retain all of its old meanings.

Examples:
a) The operators >> and << may be used for I/O operations because in the
header, they are overloaded.

b) In a stack class, it is possible to overload the + operator so that it appends
the contents of one stack to the contents of another. But the + operator still
retains its original meaning relative to other types of data.

2) The following rules constrain how overloaded operators are implemented.
However, they do not apply to the new and delete operators, which are
covered separately.

You cannot define new operators, such as **.

You cannot redefine the meaning of operators when applied to built-in data types.

Overloaded operators must either be a non-static class member function or a global function.

Operators obey the precedence, grouping, and number of operands dictated by their typical use with built-in types.

Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.

Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.

If an operator can be used as either a unary or a binary operator (&, *, +, and -), you can overload each use separately.

Overloaded operators cannot have default arguments.

All overloaded operators except assignment (operator=) are inherited by derived classes.

The first argument for member-function overloaded operators is always of the class type of the object for which the operator is invoked (the class in which the operator is declared, or a class derived from that class). No conversions are supplied for the first argument.

3) Function overloading is like you have different functions with the same name but different signatures working differently. So, the compiler can differentiate and find out which function to call depending on the context. In case of operator overloading, you try to create your own functions which are called when the corresponding operator is invoked for the operands.

One important thing to understand is that you can create as many functions as you want with the same name and different signatures so that they can work differently but for a particular class, you cannot overload the operator function based on number of arguments. There is a fundamental reason behind this.

According to the rules, you can not create your own operators and you have to use already available operators. Another thing is, since the operators are already defined for use with built-in types, you can't change their characteristics. For example, the binary operator '+' always takes two parameters, so for this you cannot create a function that takes three parameters. But you can always overload them based on the type of the parameters.

4) Multiple Choice Questions

   i)   (D)

   ii) (C)

   iii) (D)

   iv) (A)

2.9 FURTHER READINGS

1) *Object-Oriented Programming with C++*, E. Balagurusamy, 2nd edition, TMH, New Delhi, 2001

2) *The C++ Programming Language*, Bjarne Stroustroup, 3rd edition, Addison Wesley, 1997

3) *C++: The Complete Reference,* H. Shildt, 4th edition, TMH, New Delhi, 2004

4) *Mastering C++,* K. R. Venugopal, Rajkumar and T. Ravishankar, TMH, New Delhi, 2004

5) www.learncpp.com/cpp-tutorial

6) http://www.bogotobogo.com/cplusplus