
MODULE - III

Operator Overloading and Inheritance

Operator Overloading and Inheritance

Module Description

The main goal of studying operator overloading in C++ is to understand the how those operators can be overloaded. This module explains the Operator overloading for unary and binary operators and data type conversions in C++.

By the end of this module, students will learn how to overload different operators. The students will also be equipped to write some sample programs using overloading of operators. In addition to these skills students are also able to intelligently compile and execute the program and debug the errors.

Chapter 3.1

Operator Overloading

Chapter 3.2

Inheritance

Chapter Table of Contents

Chapter 3.1

Operator Overloading

Aim.....	147
Instructional Objectives.....	147
Learning Outcomes.....	147
3.1.1 Introduction.....	148
3.1.2 Operator Overloading	148
(i) Defining Operator Overloading	148
(ii) Overloading Unary Operator	150
(iii) Overloading Binary Operators.....	157
(iv) Manipulation of String Using Overloaded Operator.....	159
(v) Rules for Overloading Operator	160
Self-assessment Questions.....	161
3.1.3 Data Conversion.....	162
(i) Conversion between Basic Types	162
(ii) Conversion between Objects and Basic Types	164
(iii) Conversions between Objects of Different Classes.....	167
(iv) Basis for the Holistic	169
Self-assessment Questions.....	170
Summary	171
Terminal Questions.....	171
Answer Keys.....	172
Activity.....	172
Bibliography.....	173
e-References	173
External Resources	173
Video Links	173



Aim

To equip the students to write sample programs using operator overloading in C++ programming



Instructional Objectives

After completing this chapter, you should be able to:

- Explain operator overloading for unary and binary operators
- Illustrate Manipulation of string using overloaded operator
- Explain data conversion



Learning Outcomes

At the end of this chapter, you are expected to:

- Discuss the rules for operator overloading
- Design a program to demonstrate operator overloading
- Write a code for overloading of Unary and binary operator
- Demonstrate type conversion in C++
- Explain conversion between objects of different classes

3.1.1 Introduction

In the previous chapter, we came across the basic concepts of classes and objects. That is, how objects can be used in order to access the members belonging to a particular class. In this chapter, we will study how the functionality of classes and objects can be further extended in operator overloading which forms an important feature in C++ programming.

The most exciting feature of C++ is Operator overloading. The predefined operators such as +, =, *, /, >, < etc., are useful in any programming language. The programmers can use these operators directly on built-in data types to write programs. However, these operators do not work for user-defined types such as objects. Thus, in C++ when programmers operate on class objects, it is necessary to redefine the meaning of operators. This feature is called operator overloading. With this feature, the overloading principle is applied to functions as well as operators.

In this chapter, we will study the basic concepts of operator overloading and its use in C++ programming. Further, we will also discuss the different types of operators such as unary and binary operators that can be overloaded.

3.1.2 Operator Overloading

It can transform complex, obscure program listings into intuitively obvious ones. To make the program clearer, programmers operate on operators.

Example 1: Statements like `a3.addobjects (a1, a2);` or `a3 = a1.addobjects (a2);` can be changed to the much more readable `a3 = a1 + a2;`

Example 2: To perform multiplication on different data types '*' operator can be overloaded. The Statements like `result = mul (add(x, y), div(x, y));` can be replaced with `result = x + y * x / y;` which is easy to understand, clearer and more readable.

(i) Defining Operator Overloading

A specific case of polymorphism is operator overloading (less commonly known as ad-hoc polymorphism) is (part of the OO nature of the language) in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviours depending on the types of its arguments.

When an operator has to be overloaded, it is declared in the public section. The syntax for an operator overloading function is as follows.

```
Class classname
{
    public:
    return_type operator op(arguments if any)
        { -----
          //Function body
          -----
        }
    -----
};
```

Here operator is a keyword and op is the operator (++ , -- , ! , -) that has to be overloaded.

Example:

```
void operator ++()
{
    count = count+1;
}
```

Here void is the return_type and ++ is an op that has to be overloaded.

Operator overloading can be implemented in 2 ways as follows:

1. Using member function
2. Using friend function

Although operators can be easily overloaded using any of these techniques, its programmers convenience to choose the technique. Let us see in table 3.1.1 the differences between operator overloading using member function and friend function.

Table 3.1.1: Difference between Operator Overloading using Member Function and Friend Function

Member function	Friend function
As the calling object is implicitly supplied as an operand, the number of explicit parameters is reduced by one	The number of explicit parameters is more

Unary operator takes no explicit parameters	Unary operator take only one parameter
Binary operators take one explicit parameter	Binary operators take two parameters
Left-hand operand has to be the calling object.	Left-hand operand need not be an object of the class
ob2 = ob1 + 20; is permissible but ob2 = 20 + ob1; is not permissible	ob2 = ob1 + 20; as well as ob2 = 20 + ob1; is permissible

(ii) Overloading Unary Operator

Class is an abstract data type. The core idea of class in object oriented programming is to Placing data and function together into a single entity.

Unary operators act on only one operand. (An operand is simply a variable acted on by an operator.)

The following are some of the examples unary operators.

- Increment operator (++)
- Decrement operator (--)
- Unary minus operator (-)
- Logical not operator (!)

Assume, we have a member function,

```
a1.inc_count();
```

We call this member function to increment the count by 1. The above statement also does the job. But the listing would have been more readable if we could have used the increment operator ++ instead:

```
++a1;
```

Using a member function to overload a unary operator: The syntax for operator overloading using a member function is

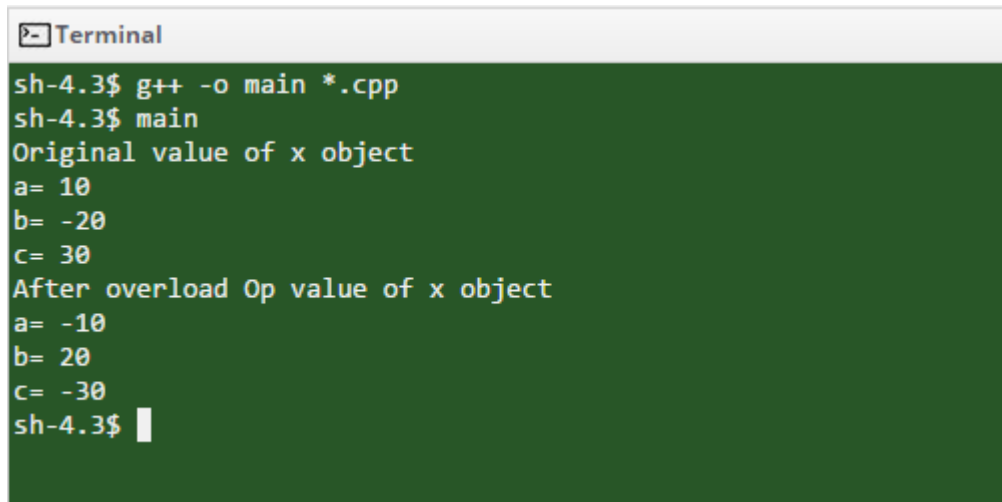
```
return_type operator op ( )
```

Where return_type is the return type and op is the operator to be overloaded. Let us consider a small program that overloads the unary operator using member function.

For example,

```
1  #include<iostream>
2  using namespace std;
3
4  class abc
5  {
6  private:
7
8      int a;
9      int b;
10     int c;
11 public:
12     abc()
13     {
14         a=10;
15         b=-20;
16         c=30;
17     }
18     void operator -();
19     void display()
20     {
21         cout<<"a= "<<a<<endl;
22         cout<<"b= "<<b<<endl;
23         cout<<"c= "<<c<<endl;
24     }
25 };
26 void abc::operator -()
27 {
28     a=-a;
29     b=-b;
30     c=-c;
31 }
32 int main()
33 {
34     abc x; // x object created and constructor executed
35     cout<<"Original value of x object"<<endl;
36     x.display();
37     -x; // Operator overloading function is execute
38     cout<<"After overload Op value of x object"<<endl;
39     x.display();
40 }
41 }
```

Output:

A terminal window titled "Terminal" with a dark green background and white text. It shows the compilation of a C++ program using g++ and its execution. The program prints the original values of variables a, b, and c, and then prints their values after an overloaded unary operator has been applied.

```
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Original value of x object
a= 10
b= -20
c= 30
After overload Op value of x object
a= -10
b= 20
c= -30
sh-4.3$
```

Using a friend function to overload a unary operator: The syntax for operator overloading using a friend function is

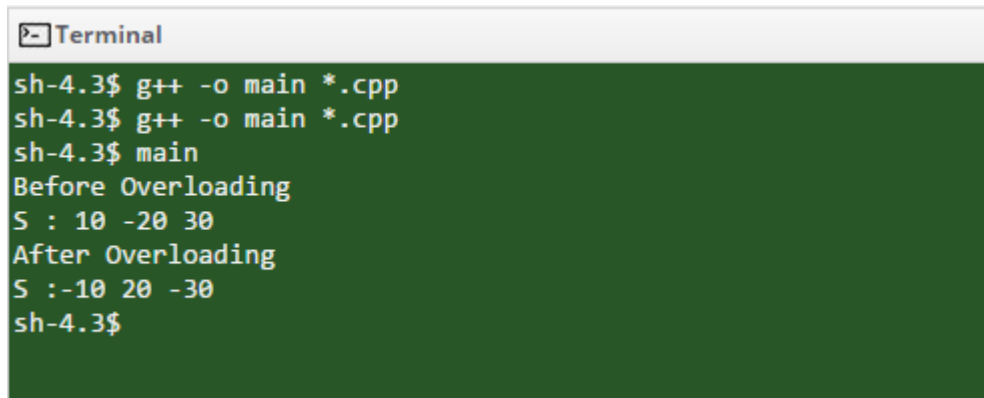
```
friend return_type operator op (class_name object)
```

Where return_type is the return type and op is the operator to be overloaded and object is an instance of a class on which the operator had to be applied. The program below performs the same operation as the previous example but using a friend function.

For example,

```
1  #include<iostream>
2  using namespace std;
3
4  class space{
5      int x;
6      int y;
7      int z;
8      public:
9          void getdata(int a,int b,int c);
10         void display(void);
11         friend void operator-(space &s);
12     };
13     void space :: getdata(int a,int b,int c){
14         x=a;
15         y=b;
16         z=c;
17     }
18     void space :: display(void){
19         cout<<x<<" ";
20         cout<<y<<" ";
21         cout<<z<<"\n";
22     }
23     void operator-( space &s){
24         s.x=-s.x;
25         s.y=-s.y;
26         s.z=-s.z;
27     }
28     int main()
29     {
30         space s;
31         s.getdata(10,-20,30);
32         cout<<"Before Overloading\n";
33         cout<<"S : ";
34         s.display();
35         -s;
36         cout<<"After Overloading\n";
37         cout<<"S : ";
38         s.display();
39         return 0;
40     }
```

Output:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Before Overloading
S : 10 -20 30
After Overloading
S :-10 20 -30
sh-4.3$
```

Overloading the Prefix Increment and Decrement Operators: The syntax for overloading prefix increment and decrement operators is

```
operator ++()
{
    //code
}
```

In the below example, we have overloaded ++ and – operators, while ++ operator has been overloaded using a member function, the – operator on the other hand has been overloaded using a friend function. Although operators can be overloaded in any of the 2 ways, we have used both the techniques for better clarity. Note that the function code for both the operators first changes the original value and then returns the object with the modified value.

```

1  #include<iostream>
2  using namespace std;
3
4  class Num
5  {
6  private:
7  int num1;
8  public:
9  Num(int num2=0) : num1(num2)
10 {
11 }
12 Num& operator++();
13 Num& operator--();
14 friend std::ostream& operator<< (std::ostream &out, const Num &n);
15 };
16
17 Num& Num::operator++()
18 {
19     // If our number is already at 9, wrap around to 0
20     if (num1 == 9)
21         num1 = 0;
22     // otherwise just increment to next number
23     else
24         ++num1;
25
26     return *this;
27 }
28
29 Num& Num::operator--()
30 {
31     // If our number is already at 0, wrap around to 9
32     if (num1 == 0)
33         num1 = 9;
34     // otherwise just decrement to next number
35     else
36         --num1;
37     return *this;
38 }
39
40 std::ostream& operator<< (std::ostream &out, const Num &n)
41 {
42     out<< n.num1;
43     return out;
44 }
45
46 int main()
47 {
48     Num num2(8);
49     std::cout << num2;
50     std::cout << ++num2;
51     std::cout << ++num2;
52     std::cout << --num2;
53     std::cout << --num2<<endl;
54     return 0;
55 }

```

Overloading the Postfix Increment and Decrement Operators: Although, the prefix and postfix increment operators both has the same symbol ++, they have different tasks to perform. To distinguish between the overloaded prefix and the postfix operator function, there is a slight change in the syntax of postfix operation. Instead of writing

Num operator ++ (), we write Num operator ++(int).

Let us look at the program which overloads both the prefix and postfix increment as well as decrement operators.

For example,

```
1  #include<iostream>
2  using namespace std;
3  class pp
4  {
5      int a,b;
6      public:
7          pp()
8          {
9              a=5;
10             b=-5;
11         }
12         void show()
13         {
14             cout<<" a="<<a<<" and b="<<b;
15         }
16     }
17
18     void operator ++()
19     {
20         ++a;
21         ++b;
22     }
23     void operator ++(int)
24     {
25         a++;
26         b++;
27     }
28     void operator --()
29     {
30         --a;
31         --b;
32     }
33     void operator --(int)
34     {
35         a--;
36         b--;
37     }
38 };
```

```

39
40 int main()
41 {
42     pp p,p1,p2,p3;
43     cout<<"Before prefix increment";
44     p.show();
45     ++p;
46     cout<<"\nAfter prefix increment";
47     p.show();
48     cout<<"\nBefore postfix increment";
49     p1.show();
50     p1--;
51     cout<<"\nAfter postfix increment";
52     p1.show();
53
54     cout<<"\n\nBefore prefix decrement";
55     p2.show();
56     --p2;
57     cout<<"\nAfter prefix decrement";
58     p2.show();
59     cout<<"\nBefore postfix decrement";
60     p3.show();
61     p3--;
62     cout<<"\nAfter postfix decrement";
63     p3.show();
64
65 }

```

Output:

```

sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Before prefix increment a=5 and b=-5
After prefix increment a=6 and b=-4
Before postfix increment a=5 and b=-5
After postfix increment a=4 and b=-6

Before prefix decrement a=5 and b=-5
After prefix decrement a=4 and b=-6
Before postfix decrement a=5 and b=-5
After postfix decrement a=4 and b=-6sh-4.3$

```

(iii) Overloading Binary Operators

The binary operators take two arguments, we use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator. Following example explains how addition (+) operator can be overloaded. Similar way, we can overload subtraction (-) and division (/) operators.

For example,

```
1  #include <iostream>
2  using namespace std;
3
4  class Box
5  {
6      double length;    // Length of a box
7      double breadth;   // Breadth of a box
8      double height;    // Height of a box
9  public:
10
11      double getVolume(void)
12      {
13          return length * breadth * height;
14      }
15      void setLength( double len )
16      {
17          length = len;
18      }
19
20      void setBreadth( double bre )
21      {
22          breadth = bre;
23      }
24
25      void setHeight( double hei )
26      {
27          height = hei;
28      }
29      // Overload + operator to add two Box objects.
30      Box operator+(const Box& b)
31      {
32          Box box;
33          box.length = this->length + b.length;
34          box.breadth = this->breadth + b.breadth;
35          box.height = this->height + b.height;
36          return box;
37      }
38 };
39 // Main function for the program
40 int main( )
41 {
42     Box Box1;           // Declare Box1 of type Box
43     Box Box2;           // Declare Box2 of type Box
44     Box Box3;           // Declare Box3 of type Box
45     double volume = 0.0; // Store the volume of a box here
46
47     // box 1 specification
48     Box1.setLength(8.0);
49     Box1.setBreadth(9.0);
50     Box1.setHeight(10.0);
51 }
```

```

52 // box 2 specification
53 Box2.setLength(11.0);
54 Box2.setBreadth(12.0);
55 Box2.setHeight(15.0);
56
57 // volume of box 1
58 volume = Box1.getVolume();
59 cout << "Volume of Box1 : " << volume << endl;
60
61 // volume of box 2
62 volume = Box2.getVolume();
63 cout << "Volume of Box2 : " << volume << endl;
64
65 // Add two object as follows:
66 Box3 = Box1 + Box2;
67
68 // volume of box 3
69 volume = Box3.getVolume();
70 cout << "Volume of Box3 : " << volume << endl;
71
72 return 0;
73 }

```

Output:

```

Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
Volume of Box1 : 720
Volume of Box2 : 1980
Volume of Box3 : 9975
sh-4.3$

```

(iv) Manipulation of String Using Overloaded Operator

C++ allows us the facility of manipulating strings using the concept of operator overloading. We can overload + operator to concatenate 2 strings and we can overload == operator to compare 2 strings.

The below example program is in which we overload + operator to concatenate two strings

```

#include<conio.h>
#include<string.h>
#include<iostream.h>
class string
{
public:
char *str;
int size;
void getstring(char *str)
{

```

```

        size = strlen(s);
    str = new char[size];
    strcpy(str,s);
    }
void operator+(string);
};
void string::operator+(string obj)
{
    size = size+obj.size;
    str = new char[size];
    strcat(str,obj.str);
    cout<<"\n Concatenated String is: "<<str;
}
void main()
{
    string obj1, obj2;
    char *str1, *str2;
    clrscr();
    cout<<"\nEnter First String:";
    cin>>str1;
    obj1.getstring(str1);
    cout<<"\nEnter Second String:";
    cin>>str2;
    obj2.getstring(str2);
    //Calling + operator to Join/Concatenate strings
    obj1+obj2;
    getch();
}

```

Output:

Enter first string: Objectoriented

Enter second string: programming

Concatenated String is Objectorientedprogramming

(v) Rules for Overloading Operator

- The existing operators can only be overloaded. New operators can't be made
 - The overloaded operator must have at least one operand that is of user-created type.
 - The essential importance of an operator can't be changed. That is addition operator can't be utilized to subtract one number from the other.
 - Overloaded operator follows the guidelines of existing operators. Hence they are not overridden.
 - There are a few operators that can't be overloaded. They are Size of, ., :, ;, ?..
-

-
- Friend function can't be utilized to over-burden certain operators (= , () , [] , - >). However part functions can be utilized to over-burden them.
 - Unary operators, overload by method for a member function, take no explicit arguments and give back no explicit values but those overloaded by method for a friend function, take one reference argument.
 - Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
 - At the point when utilizing binary operator overloaded through a member function, the left hand operand must be an object of the significant class.
 - Binary math operators, *for example*, +, -, *, and / should explicitly give back an value. They should not try to change their own arguments.



Self-assessment Questions

- 1) Operators can be refined using _____ concepts.
 - 2) Operator overloading implements the concept of polymorphism.
 - a) True
 - b) False
 - 3) Choose the operator which can be overloaded.
 - a) ::
 - b) ++
 - c) **
 - d) .*
 - 4) Operator overloading cannot change the operation performed by an operator?
 - a) True
 - b) False
 - 5) With friend function implementation, binary operator takes how many explicit parameters.
 - a) 0
 - b) 3
 - c) 1
 - d) 2
-

3.1.3 Data Conversion

The process of converting a value of data type in to another data type is known as data conversion or data type conversion.

For example,

```
int no1,no2;  
no1=no2;
```

Here, the compiler automatically converts one data type to another by the use of '=' sign between the two variables, no1 and no2.

(i) Conversion between Basic Types

The scope resolution operator (::) can be used to access data members outside the class passed to a member function of an object in a class.

This is a straight forward process and is frequently done implicitly.

For example,

```
int n1;  
float n2 = 0.16;
```

Now, If we want to assign the value of float to n1, we can do as shown in the below statement.

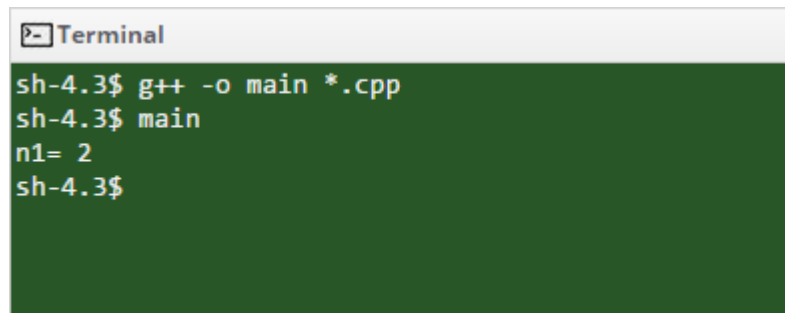
```
n1= n2;
```

In this case, the Compiler will not generate any error though; n1 is of integer type whereas n2 is of float type. This is implicit conversion where compiler will run some internal routine to convert the float value to integer. However, we can also direct the compiler to convert the float type to integer. This is called explicit conversion of basic type data. In the below example, we have explained how compiler implicitly converts basic data types.

For example,

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7      int n1;
8      float n2 = 2.165;
9      n1=n2;
10     cout<<"n1= "<<n1<<endl;
11 }
12
```

Output:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
n1= 2
sh-4.3$
```

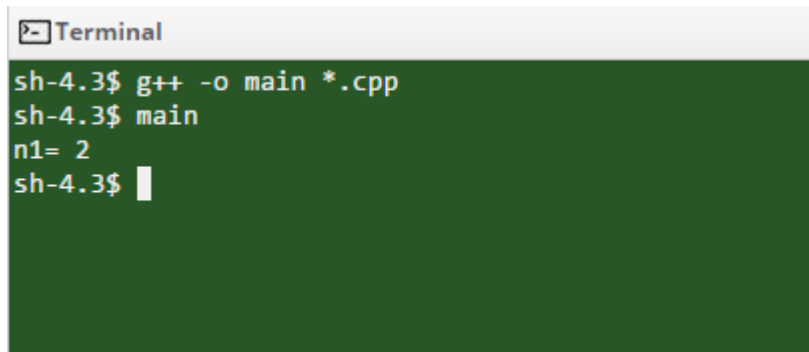
Here we have two variables, n1 of type integer and n2 of type float and we initialized float with value 2.165. Then we stored the value of float variable in integer variable. We simple assigned the value using equals sign. We did not provide any programming logic, rather the compiler automatically run the conversion routine and store the integer type value of float into integer variable.

In the above *Example*, float was implicitly converted into integer; truncating the decimal part. However, for the purpose of readability, we can also explicitly specify the conversion between float and integer. Look at below *Example*.

For example,

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7      int n1;
8      float n2 = 2.165;
9      n1=static_cast<int>(n2);
10     cout<<"n1= "<<n1<<endl;
11 }
12
```

Output:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
n1= 2
sh-4.3$
```

In the above Example, we have explicitly converted float into integer using the statement

```
n1=static_cast<int>(n2);
```

The above statement shows that n2 is being converted into integer type. If we want to convert it into any other type, we can simply replace <int> with the <type>, where type is the type of data into which you want the conversion to be done.

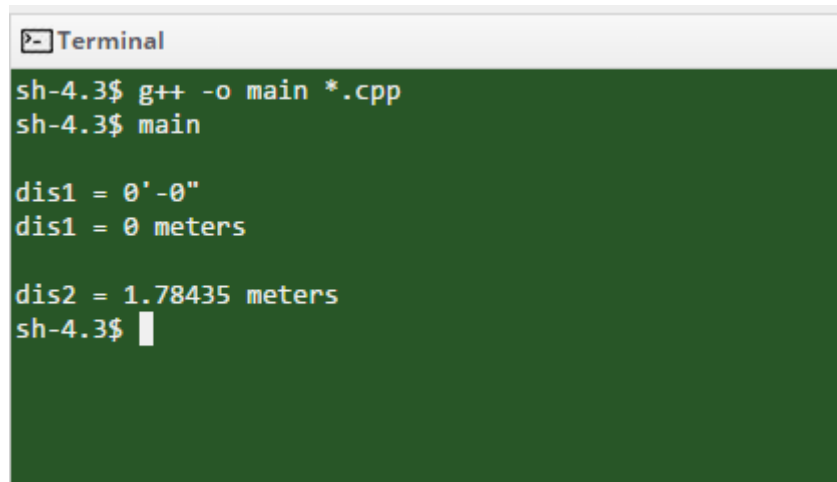
(ii) Conversion between Objects and Basic Types

We cannot depend or rely on built-in conversion routines to convert between user-defined data types and basic data types. The compiler will not understand the user-defined types, so we must write user-defined routines ourselves.

// conversions: Distance to meters, meters to Distance

```
1  #include <iostream>
2  using namespace std;
3  class Dis //English Distance class
4  {
5  private:
6  const float MF; //meters to feet
7  int ft;
8  float in;
9  public: //constructor (no args)
10 Dis() : ft(0), in(0.0), MF(3.280833F)
11 { }
12 Dis(float mt) : MF(3.280833F)
13 { //convert meters to Distance
14 int meters;
15 float fltft = MF * meters; //convert to float feet
16 ft = int(fltft); //feet is integer part
17 in = 12*(fltft-ft); //inches is what's left
18 } //constructor (two args)
19
20
21 Dis(int f, float i) : ft(f),
22 in(i), MF(3.280833F)
23 { }
24 void getdist() //get length from user
25 {
26 cout<< "\nEnter feet: "; cin >>ft;
27 cout<< "\nEnter inches: "; cin >> in;
28 }
29 void showdist() const //display distance
30 { cout<<ft<< "\'-" << in<< '\n'; }
31 operator float() const //conversion operator
32 { //converts Distance to meters
33 float fracft = in/12; //convert the inches
34 fracft += static_cast<float>(ft); //add the feet
35 return fracft/MF; //convert to meters
36 }
37 };
38
39 int main()
40 {
41 float mts;
42 Dis dis1 = 2.35F; //uses 1-arg constructor to
43 //convert meters to Distance
44 cout<< "\ndis1 = "; dis1.showdist();
45 mts = static_cast<float>(dis1); //uses conversion operator
46 //for Distance to meters
47 cout<< "\ndis1 = " <<mts<< " meters\n";
48 Dis dis2(5, 10.25); //uses 2-arg constructor
49 mts = dis2; //also uses conversion op
50 cout<< "\ndis2 = " <<mts<< " meters\n";
51 // dis2 = mts; //error, = won't convert
52 return 0;
53 }
```

Output:

A terminal window titled "Terminal" with a dark green background. It shows the following commands and output:

```
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main

dis1 = 0'-0"
dis1 = 0 meters

dis2 = 1.78435 meters
sh-4.3$
```

In `main()` the program first converts a fixed float quantity—2.35, representing meters—to feet and inches, using the one-argument constructor: `Dis dis1 = 2.35F`; The statements `mtrs = static_cast<float>(dist2);` and `mts = dis2;` converts a `Distance` to meters in the statements.

Let's discuss more about what happens in the `Distance` member function. Converting a user-defined type to a basic type requires a different approach than converting a basic type to a user-defined type.

From Basic to User-Defined

Conversion constructors are used to convert from a basic type say float to a user-defined type such as `Dis`. These constructors are constructors with only one argument and they not declared with the 'explicit' specifier.

```
Dis(float mts)
{
    float fltft = MTF * mts;
    ft = int(fltft);
    in = 12 * (fltft-ft);
}
```

This function is called when an object is created with a single argument of type `Dis`. The function assumes that this argument represents meters. It converts the argument to feet and inches and assigns the resulting values to the object. Thus the conversion from meters to `Distance` is carried out along with the creation of an object in the statement

```
Dis dis1 = 2.35;
```

Conversion from User-Defined to Basic Type

We can convert from a user-defined type to a basic type using Conversion Operator. Consider the below example:

```
operator float()
{
    floatfracft = in/12;
    fracft += float(ft);
    returnfracft/MF;
}
```

This operator takes the value of the Dis object of which it is a member, converts it to a float value representing meters and returns this value.

This operator can be called with an explicit cast

```
mts = static_cast<float>(dis1);
```

Or with a simple assignment

```
mtrs = dis2;
```

Both forms convert the Distance object to its equivalent float value in meters.

(iii) Conversions between Objects of Different Classes

Using one argument instructor, the conversion between objects of different class can be done. We can also use a conversion function. The usage of this depends on whether we want to use the conversion routine in the class specifier of the source object or of the destination object.

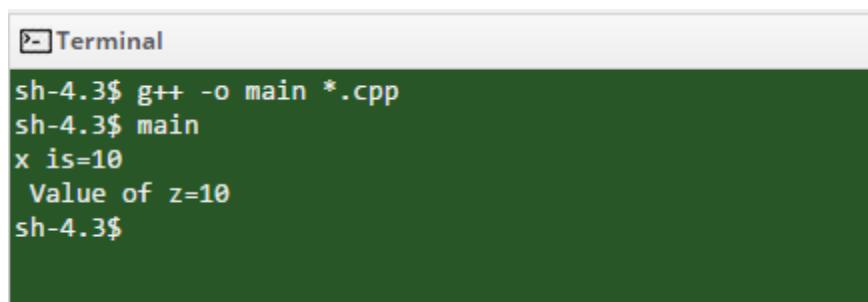
Routine in Source object

The conversion routine might be single-argument constructor or an operator function. For user-defined objects in case of source objects, operator function of source class is used.

The *example* program below shows a conversion routine in the source class.

```
1  #include<iostream>
2  using namespace std;
3
4  class sum
5  {
6  private:
7  int x,y;
8  public:
9  sum(int a)
10 {
11     x=a;
12 }
13
14 operator int()
15 {
16     return x;
17 }
18
19 void show()
20 {
21     cout<<"x is="<<x;
22 }
23 };
24
25 int main()
26 {
27     int z;
28     sum s1(10);
29     s1.show();
30     z=s1;
31     cout<<"\n Value of z="<<z<<endl;
32 }
```

Output:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
x is=10
Value of z=10
sh-4.3$
```

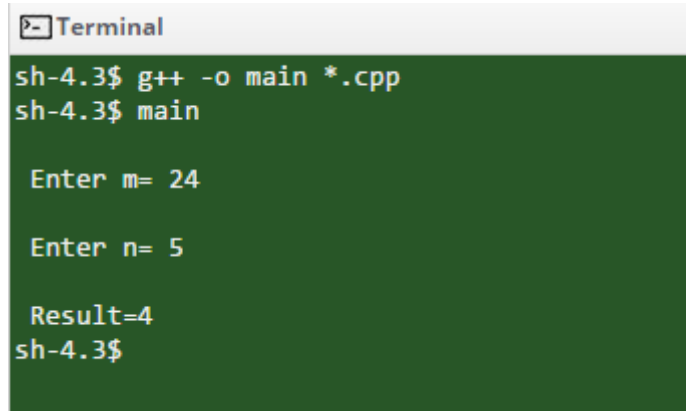
Routine in Destination Object

The conversion routine might be single-argument constructor or an operator function. For user-defined objects in case of destination objects, single argument constructor of destination class is used.

The **example** program below shows a conversion routine in the destination.

```
1  #include<iostream>
2  using namespace std;
3
4  class sum
5  {
6  private:
7  float x,y;
8  public:
9  void getdata(float a,float b)
10 {
11     x=a;
12     y=b;
13 }
14 void display()
15 {
16     float z;
17     z=int(x/y);
18     cout<<"\n Result="<<z;
19 }
20 };
21
22 int main()
23 {
24     sum s1;
25     float m,n,z;
26     cout<<"\n Enter m= ";
27     cin>>m;
28     cout<<"\n Enter n= ";
29     cin>>n;
30     s1.getdata(m,n);
31     s1.display();
32 }
```

Output:



```
Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main

Enter m= 24

Enter n= 5

Result=4
sh-4.3$
```

(iv) Basis for the Holistic

The below program illustrates data conversion of user defined data types using functions. In the below example program the string data type is converted to integer data type.

```

1  #include <iostream>
2  #include <sstream>
3  using namespace std;
4  int main()
5  {
6  int num_int=123;
7  float num_float=123.456;
8  char num_str1[10],num_str2[10];
9  stringstream test(stringstream::in | stringstream::out);
10 test<<num_int<<endl<<num_float;    //output to the stringstream
11 test>>num_str1>>num_str2;    //input from the stringstream to the character arrays
12 cout<<num_str1<<endl<<num_str2<<endl;
13 return 0;
14 }

```

Output:

```

Terminal
sh-4.3$ g++ -o main *.cpp
sh-4.3$ main
123
123.456
sh-4.3$ 

```



Did you know?!

Overloaded -> operator function must be a non-static member function of the class.



Self-assessment Questions

- 6) When type conversion is performed in destination class, which among the following is used
 - a) Data members
 - b) Member function
 - c) Destructor
 - d) Constructor
- 7) A member function is used to convert data from one class to another
 - a) True
 - b) False
- 8) The type casting function does not have any _____.
 - a) Return type
 - b) Data type
 - c) Function
 - d) Class



Summary

- Using operator overloading, we came across compile time polymorphism which makes our programs to behave differently depending on the type of arguments that are passed in the call.
- Operator overloading can be performed either on unary operators (one operator) or binary operators (two operators).
- Also, we can perform manipulation of strings by the use of operator overloading.
- In data conversion, we saw how a value can be converted from one data type to another data type such as conversion from basic data type to user defined data type or from class type to basic data type or from one class type to another class type.
- The conversion routine can be a single-argument constructor or an operator function depending on the type of object (source or destination) respectively.



Terminal Questions

1. Explain operator overloading for unary and binary operators.
2. Explain overloading unary operator using a friend function.
3. Explain overloading unary operator using a member function.
4. Illustrate Manipulation of string using overloaded operator.
5. Explain data conversion.



Answer Keys

Self-assessment Questions	
Question No.	Answer
1	Operator Overloading
2	True
3	b
4	b
5	a
6	d
7	d
8	False



Activity

Activity Type: Online

Duration: 30 Minutes

Description:

Design and execute a program that substitutes an overloaded += operator for the overloaded + operator in the STRPLUS program in this chapter. This operator should allow statements like `s1 += s2`; where `s2` is added (concatenated) to `s1` and the result is left in `s1`. The operator should also permit the results of the operation to be used in other calculations, as in `s3 = s1 += s2`;

Bibliography



e-References

- studytonight.com (2016). *Operator Overloading in C++*. Retrieved 7 April, 2016. from, <http://www.studytonight.com/cpp/operator-overloading.php>
- cplusplus.com,(2016). *Type conversions - C++*. Retrieved 7 April, 2016. from, <http://www.cplusplus.com/doc/tutorial/typecasting/>



External Resources

- Balaguruswamy, E. (2008). *Object Oriented Programming with C++*. Tata McGraw Hill Publications.
- Lippman. (2006). *C++ Primer* (3rd ed.). Pearson Education.
- Robert, L. (2006). *Object Oriented Programming in C++* (3rd ed.). Galgotia Publications References.
- Schildt, H., & Kanetkar, Y. (2010). *C++ completer* (1st ed.). Tata McGraw Hill.
- Strousstrup. (2005). *The C++ Programming Language* (3rd ed.). Pearson Publications.



Video Links

Topic	Link
Operator overloading	https://www.youtube.com/watch?v=PgGhEovFhd0
Data conversion	https://www.youtube.com/watch?v=w2FJHN3T5C0
Rules for Operator Overloading in C++	https://www.youtube.com/watch?v=Vtq5AZLCdQw



Notes:

