Chapter 5 : Operator Overloading

Overloading Binary Operators

Overloading Binary Operator : In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands.

Let's take the same example of class Distance, but this time, add two distance objects.

```
// C++ program to show binary operator overloading
#include <iostream>
using namespace std;
```

```
class Distance {
public:
    // Member Object
    int feet, inch;
 // No Parameter Constructor
    Distance()
    {
        this->feet = 0;
        this->inch = 0;
    }
```

```
Distance(int f, int i) // Constructor to initialize the object's value
Parameterized Constructor
 {
      this->feet = f;
      this->inch = i;
   }
// Overloading (+) operator to perform addition of two distance object
   // Distance operator+(Distance& d2) // Call by reference
 {
Distance d3; // Create an object to return
```

```
d3.feet = this->feet + d2.feet;   // Perform addition of feet and inches
d3.inch = this->inch + d2.inch;  // Return the resulting object
        return d3;
    }
};
int main() // Driver Code
{
    Distance d1(8, 9);  // Declaring and Initializing first object
    Distance d2(10, 2);  // Declaring and Initializing second object
    Distance d3;  // Declaring third object
```

```
  d3 = d1 + d2; // Use overloaded operator
// Display the result
cout << "\nTotal Feet & Inches : " << d3.feet << "'" << d3.inch;
    return 0;
 }
```

Output :

```
Total Feet & Inches : 18'11
```

Overloading Unary Operators

Overloading Unary Operator : Let us consider to overload (-) unary operator. In unary operator function, no arguments should be passed. It works only with one class objects. It is a overloading of an operator operating on a single operand.

Example :

Assume that class Distance takes two member object i.e. feet and inches, create a function by which Distance object should decrement the value of feet and inches by 1 (having single operand of Distance Type).

// **Write a C++ program to show unary operator overloading**

```
#include <iostream>
using namespace std;
class Distance {
public:
    int feet, inch;  // Member Object
 Distance(int f, int i)  // Constructor to initialize the object's value
    {
        this->feet = f;
        this->inch = i;
 }
```

```
void operator-() // Overloading(-) operator to perform decrement operation
of Distance object
{
        feet--;
        inch--;
        cout << "\nFeet & Inches(Decrement) : " << feet << "'" << inch;
    }
  };
int main()   // Driver Code
{
```

```
Distance d1(8, 9); // Declare and Initialize the constructor
-d1;  // Use (-) unary operator by single operand
    return 0;
 }
```

Output :

```
Feet & Inches(Decrement) : 7'8
```

Overloading Binary Operator using a Friend function

In this approach, the operator overloading function must precede with friend keyword, and declare a function class scope. Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator.

All the working and implementation would same as binary operator function except this function will be implemented outside of the class scope.

Let’s take the same example using the friend function.

// Write a C++ program to show binary operator overloading

```
#include <iostream>
using namespace std;
class Distance {
public :
    int feet, inch; // Member Object
    Distance()   // No Parameter Constructor
 {
        this->feet = 0;
        this->inch = 0;
    }  // Constructor to initialize the object's value
```

```
 Distance(int f, int i)  // Parameterized Constructor
    {
        this->feet = f;
        this->inch = i;
    }
    friend Distance operator+(Distance&, Distance&);
// Declaring friend function using friend keyword
};
   // Implementing friend function with two parameters
Distance operator+(Distance& d1, Distance& d2) // Call by reference
{
   Distance d3;  // Create an object to return
```

```
d3.feet = d1.feet + d2.feet; // Perform addition of feet and inches
 d3.inch = d1.inch + d2.inch;
    return d3; // Return the resulting object
}
int main()  // Driver Code
{
    Distance d1(8, 9); // Declaring and Initializing first object
    Distance d2(10, 2);  // Declaring and Initializing second object
   Distance d3;   // Declaring third object
   d3 = d1 + d2;  // Use overloaded operator
  // Display the result
  cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
```

```
    return 0;
}
```

Rules for Overloading Operators

In C++, following are the general rules for operator overloading.

1. Only built-in operators can be overloaded. New operators can not be created.

2. Arity of the operators cannot be changed.

3. Precedence and associativity of the operators cannot be changed.

4. Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.

5. Operators cannot be overloaded for built in types only. At least one operand must be used defined type.

6. Assignment (=), subscript ([]), function call (“()”), and member selection (->) operators must be defined as member functions.

7. Except the operators specified in point 6, all other operators can be either member functions or a non- member functions.

8. Some operators like (assignment)=, (address)& and comma (,) are by default overloaded.

Types of Converions

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. Implicit Type Conversion Also known as ‘automatic type conversion’ : Done by the compiler on its own, without any external trigger from the user.

- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion :

```
// An example of implicit conversion
#include <iostream>
using namespace std;
int main()
{
```

```
int x = 10; // integer x
 char y = 'a'; // character c
x = x + y; // y implicitly converted to int. ASCII value of 'a' is 97
float z = x + 1.0; // x is implicitly converted to float
cout << "x = " << x << endl<< "y = " << y << endl<< "z = " << z << endl;
return 0;
 }
```

Output :

x = 107

y = a

z = 108

Explicit Type Conversion : This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type. In C++, it can be done by two ways:

1. Converting by assignment : This is done by explicitly defining th required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Syntax :

(type) expression;

Example :

```
//  Write a C++ program to demonstrate explicit type casting
#include <iostream>
using namespace std;
```

```
int main()
{
double x = 1.2;
   // Explicit conversion from double to int
    int sum = (int)x + 1;
    cout << "Sum = " << sum;
  return 0;
 }
```

Output :

```
   Sum = 2
```

2. <u>Conversion using Cast operator</u> : A Cast operator is an unary operator which forces one data type to be converted into another data type.

- C++ supports four types of casting:
- Static Cast
- Dynamic Cast
- Const Cast
- Reinterpret Cast

<u>Example</u> :

```
#include <iostream>

using namespace std;
```

```
int main()
{
    float f = 3.5;
    int b = static_cast<int>(f); // using cast operator
   cout << b;
}
```

Output :

```
 3
```

Advantages of Type Conversion

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.