Unit-6.1

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

**Operator that cannot be overloaded are as follows:**

* Scope operator (::)
* Sizeof
* member selector(.)
* member pointer selector(\*)
* ternary operator(?:)

Syntax of Operator Overloading

1. return\_type class\_name  : : operator op(argument\_list)
2. {
3. // body of the function.
4. }

Where the **return type** is the type of value returned by the function.

**class\_name** is the name of the class.

**operator op** is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

* Existing operators can only be overloaded, but the new operators cannot be overloaded.
* The overloaded operator contains atleast one operand of the user-defined data type.
* We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
* When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
* When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Test
4. {
5. **private**:
6. **int** num;
7. **public**:
8. Test(): num(8){}
9. **void** operator ++()         {
10. num = num+2;
11. }
12. **void** Print() {
13. cout<<"The Count is: "<<num;
14. }
15. };
16. **int** main()
17. {
18. Test tt;
19. ++tt;  // calling of a function "void operator ++()"
20. tt.Print();
21. **return** 0;
22. }

**Output:**

The Count is: 10

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** A
4. {
6. **int** x;
7. **public**:
8. A(){}
9. A(**int** i)
10. {
11. x=i;
12. }
13. **void** operator+(A);
14. **void** display();
15. };
17. **void** A :: operator+(A a)
18. {
20. **int** m = x+a.x;
21. cout<<"The result of the addition of two objects is : "<<m;
23. }
24. **int** main()
25. {
26. A a1(5);
27. A a2(4);
28. a1+a2;
29. **return** 0;
30. }

**Output:**

The result of the addition of two objects is : 9

# C++ Program to concatenate two strings using Operator Overloading

**Approach 1**: Using unary operator overloading.

* To concatenate two strings using unary operator overloading. Declare a class with two string variables.
* Create an instance of the class and call the Parameterized constructor of the class to initialize those two string variables with the input strings from the main function.
* Overload the unary operator to concatenate these two string variables for an instance of the class.
* Finally, call the operator function and concatenate two class variables.

// C++ Program to concatenate two string

// using unary operator overloading

#include <iostream>

#include <string.h>

using namespace std;

// Class to implement operator overloading

// function for concatenating the strings

class AddString {

public:

// Classes object of string

char s1[25], s2[25];

// Parameterized Constructor

AddString(char str1[], char str2[])

{

// Initialize the string to class object

strcpy(this->s1, str1);

strcpy(this->s2, str2);

}

// Overload Operator+ to concat the string

void operator+()

{

cout << "\nConcatenation: " << strcat(s1, s2);

}

};

// Driver Code

int main()

{

// Declaring two strings

char str1[] = "Geeks";

char str2[] = "ForGeeks";

// Declaring and initializing the class

// with above two strings

AddString a1(str1, str2);

// Call operator function

+a1;

return 0;

}

**Output:**

Concatenation: GeeksForGeeks

**Approach 2:** Using binary operator overloading.

* Declare a class with a string variable and an operator function ‘+’ that accepts an instance of the class and concatenates it’s variable with the string variable of the current instance.
* Create two instances of the class and initialize their class variables with the two input strings respectively.
* Now, use the overloaded operator(+) function to concatenate the class variable of the two instances.

Below is the implementation of the above approach:

|  |
| --- |
| // C++ Program to concatenate two strings using  // binary operator overloading  #include <iostream>  #include <string.h>    using namespace std;    // Class to implement operator overloading function  // for concatenating the strings  class AddString {    public:      // Class object of string      char str[100];        // No Parameter Constructor      AddString() {}        // Parameterized constructor to      // initialize class Variable      AddString(char str[])      {          strcpy(this->str, str);      }        // Overload Operator+ to concatenate the strings      AddString operator+(AddString& S2)      {          // Object to return the copy          // of concatenation          AddString S3;            // Use strcat() to concat two specified string          strcat(this->str, S2.str);            // Copy the string to string to be return          strcpy(S3.str, this->str);            // return the object          return S3;      }  };    // Driver Code  int main()  {      // Declaring two strings      char str1[] = "Geeks";      char str2[] = "ForGeeks";        // Declaring and initializing the class      // with above two strings      AddString a1(str1);      AddString a2(str2);      AddString a3;        // Call the operator function      a3 = a1 + a2;      cout << "Concatenation: " << a3.str;        return 0;  } |

**Output:**

Concatenation: GeeksForGeeks

# Type Conversion in C++

In this topic, we will discuss the conversion of one data type into another in the C++ programming language. Type conversion is the process that converts the predefined data type of one variable into an appropriate data type. The main idea behind type conversion is to convert two different data type variables into a single data type to solve mathematical and logical expressions easily without any data loss.

For example, we are adding two numbers, where one variable is of int type and another of float type; we need to convert or typecast the int variable into a float to make them both float data types to add them.

Type conversion can be done in two ways in C++, one is **implicit type conversion**, and the second is **explicit type conversion**. Those conversions are done by the compiler itself, called the implicit type or automatic type conversion. The conversion, which is done by the user or requires user interferences called the explicit or user define type conversion. Let's discuss the implicit and explicit type conversion in C++.

### Implicit Type Conversion

The implicit type conversion is the type of conversion done automatically by the compiler without any human effort. It means an implicit conversion automatically converts one data type into another type based on some predefined rules of the C++ compiler. Hence, it is also known as the **automatic type conversion**.

OOPs Concepts in Java

**For example:**

1. **int** x = 20;
2. **short** **int** y = 5;
3. **int** z = x + y;

In the above example, there are two different data type variables, x, and y, where x is an int type, and the y is of short int data type. And the resultant variable z is also an integer type that stores x and y variables. But the C++ compiler automatically converts the lower rank data type (short int) value into higher type (int) before resulting the sum of two numbers. Thus, it avoids the data loss, overflow, or sign loss in implicit type conversion of C++.

### Order of the typecast in implicit conversion

The following is the correct order of data types from lower rank to higher rank:

1. **bool** -> **char** -> **short** **int** -> **int** -> unsigned **int** -> **long** **int** -> unsigned **long** **int** -> **long**  **long** **int** -> **float** -> **double** -> **long** **double**

### Program to convert int to float type using implicit type conversion

Let's create a program to convert smaller rank data types into higher types using implicit type conversion.

**Program1.cpp**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main ()
4. {
5. // assign the integer value
6. **int** num1 = 25;
7. // declare a float variable
8. **float** num2;
9. // convert int value into float variable using implicit conversion
10. num2 = num1;
11. cout <<  " The value of num1 is: " << num1 << endl;
12. cout <<  " The value of num2 is: " << num2 << endl;
13. **return** 0;
14. }

**Output**

The value of num1 is: 25

The value of num2 is: 25

### Program to convert double to int data type using implicit type conversion

Let's create a program to convert the higher data type into lower type using implicit type conversion.

**Program2.cpp**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** num; // declare int type variable
6. **double** num2 = 15.25; // declare and assign the double variable
8. // use implicit type conversion to assign a double value to int variable
9. num = num2;
10. cout << " The value of the int variable is: " << num << endl;
11. cout << " The value of the double variable is: " << num2 << endl;
12. **return** 0;
13. }

**Output**

The value of the int variable is: 15

The value of the double variable is: 15.25

In the above program, we have declared num as an integer type and num2 as the double data type variable and then assigned num2 as 15.25. After this, we assign num2 value to num variable using the assignment operator. So, a C++ compiler automatically converts the double data value to the integer type before assigning it to the num variable and print the truncate value as 15.

### Explicit type conversion

Conversions that require **user intervention** to change the data type of one variable to another, is called the **explicit type conversion**. In other words, an explicit conversion allows the programmer to manually changes or typecasts the data type from one variable to another type. Hence, it is also known as typecasting. Generally, we force the explicit type conversion to convert data from one type to another because it does not follow the implicit conversion rule.

The explicit type conversion is divided into two ways:

1. Explicit conversion using the cast operator
2. Explicit conversion using the assignment operator

### Program to convert float value into int type using the cast operator

**Cast operator:** In C++ language, a cast operator is a unary operator who forcefully converts one type into another type.

Let's consider an example to convert the float data type into int type using the cast operator of the explicit conversion in C++ language.

**Program3.cpp**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main ()
4. {
5. **float** f2 = 6.7;
6. // use cast operator to convert data from one type to another
7. **int** x = **static\_cast** <**int**> (f2);
8. cout << " The value of x is: " << x;
9. **return** 0;
10. }

**Output**

The value of x is: 6

### Program to convert one data type into another using the assignment operator

Let's consider an example to convert the data type of one variable into another using the assignment operator in the C++ program.

**Program4.cpp**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main ()
4. {
5. // declare a float variable
6. **float** num2;
7. // initialize an int variable
8. **int** num1 = 25;
10. // convert data type from int to float
11. num2 = (**float**) num1;
12. cout << " The value of int num1 is: " << num1 << endl;
13. cout << " The value of float num2 is: " << num2 << endl;
14. **return** 0;
15. }

**Output**

The value of int num1 is: 25

The value of float num2 is: 25.0

**Conversion of class object to primitive data type:**

In this conversion, the **from** type is a class object and the **to** type is primitive data type. The normal form of an [overloaded casting operator](https://www.geeksforgeeks.org/operator-overloading-c/) function, also known as a conversion function. Below is the syntax for the same:

**Syntax:**

operator typename()

{

// Code

}

* Now, this function converts a **user-defined data type** to a **primitive data type**. For Example, the operator **double()** converts a class object to type double, the operator **int()** converts a class type object to type int, and so on. Below is the program to illustrate the same:

**Example:**

|  |
| --- |
| // C++ program to illustrate the  // above conversion  #include <bits/stdc++.h>  using namespace std;    // Tie Class  class Time {      int hrs, mins;    public:      // Constructor      Time(int, int);        // Casting operator      operator int();        // Destructor      ~Time()      {          cout << "Destructor is called."               << endl;      }  };    // Function that assigns value to the  // member variable of the class  Time::Time(int a, int b)  {      hrs = a;      mins = b;  }    // int() operator is used for Data  // conversion of class to primitive  Time::operator int()  {      cout << "Conversion of Class"           << " Type to Primitive Type"           << endl;        return (hrs \* 60 + mins);  }    // Function performs type conversion  // from the Time class type object  // to int data type  void TypeConversion(int hour, int mins)  {      int duration;        // Create Time Class  object      Time t(hour, mins);        // Conversion OR duration = (int)t      duration = t;      cout << "Total Minutes are "           << duration << endl;        // Conversion from Class type to      // Primitive type      cout << "2nd method operator"           << " overloading " << endl;        duration = t.operator int();        cout << "Total Minutes are "           << duration << endl;        return;  }    // Driver Code  int main()  {      // Input value      int hour, mins;      hour = 2;      mins = 20;        // Function call to illustrate      // type conversion      TypeConversion(hour, mins);        return 0;  } |

**Output**

Conversion of Class Type to Primitive Type

Total Minutes are 140

2nd method operator overloading

Conversion of Class Type to Primitive Type

Total Minutes are 140

Destructor is called.

**Conversion of one class type to another class type:**

In this type, one class type is converted into another class type. It can be done  in 2 ways :

1.Using constructor

2.Using Overloading casting operator

**1.Using constructor :**

In the Destination  class we use the constructor method

//Objects of different types

ObjectX=ObjectY;

Here ObjectX is Destination object and ObjectY is source object

**Example:**

|  |
| --- |
| #include<iostream>  using namespace std;  //cgs system  class CGS   {       int mts; //meters       int cms; //centimeters       public:         void showdata()     {         cout<<"Meters and centimeters in CGS system:";         std::cout << mts<<" meters "<<cms<<" centimeters" << std::endl;     }     CGS(int x,int y) // parameterized constructor     {         mts=x;      cms=y;     }      int getcms()     {         return cms;     }     int  getmts()     {         return mts;     }     };    class FPS   {     int feet;     int inches;     public:     FPS() // default constructor     {         feet=0;         inches=0;     }    FPS(CGS d2)     {         int x;         x=d2.getcms()+d2.getmts()\*100;         x=x/2.5;         feet=x/12;         inches=x%12;      }     void showdata()     {         cout<<"feet and inches in FPS system:";         std::cout << feet<<" feet "<<inches<<" inches" << std::endl;     }   };     int main()   {       CGS d1(9,10);       FPS d2;       d2=d1;       d1.showdata(); //to display CGS values       d2.showdata(); //to display FPS values       return 0;   } |

**Output**

Meters and centimeters in CGS system:9 meters 10 centimeters

feet and inches in FPS system:30 feet 4 inches

**2.Using Overloading casting operator**

// Objects of different types

objectX = objectY;

* Here we use Overloading casting operator in source class i.e. overloading destination class in source class

See the below example in which we have two classes Time and Minute respectively and will convert one class Time to another Minute class.

In the below example minute class is destination class and time class is source class

so we need to overload the destination class in the source class

Here we should not tell the return type but we returns the overloaded class object

i.e. returning value without specifying return type

|  |
| --- |
| // C++ program to illustrate the  // above conversion  #include <bits/stdc++.h>  using namespace std;  //minutes class  class Minute {      public:     int mins;      // Constructors      Minute()      {          mins = 0;      }        // Function to print the value of      // hours and minutes      void show()      {          cout << "\nTotal Minute : " << mins << endl;      }  };    // Time Class  class Time {      int hr, mins;    public:      // Constructors      Time(int h, int m)      {          hr = h;          mins = m;      }      Time()      {          cout << "\nTime's Object Created";      }      operator Minute  () //overloading minute class      {          Minute m;          m.mins = (hr \* 60) + mins;          return m;      } //driver code        // Function to print the value of      // hours and minutes      void show()      {          cout << "Hour: " << hr << endl;          cout << "Minute : " << mins << endl;      }  };    // Minutes Class  int main()  {      Time T1(3,40);      Minute m;      m=T1; //minute class is destination and Time class is source class      T1.show();      m.show();      return 0;  } |

**Output**

Hour: 3

Minute : 40

Total Minute : 220