Unit-4

Operator Overloading

Introduction

**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.

[**next →**](https://www.javatpoint.com/cpp-function-overriding)[**← prev**](https://www.javatpoint.com/cpp-polymorphism)

C++ Overloading (Function and Operator)

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

* methods,
* constructors, and
* indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

* Function overloading
* Operator overloading

C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Cal {
4. **public**:
5. **static** **int** add(**int** a,**int** b){
6. **return** a + b;
7. }
8. **static** **int** add(**int** a, **int** b, **int** c)
9. {
10. **return** a + b + c;
11. }
12. };
13. **int** main(**void**) {
14. Cal C;                                                    //     class object declaration.
15. cout<<C.add(10, 20)<<endl;
16. cout<<C.add(12, 20, 23);
17. **return** 0;
18. }

**Output:**

30

55

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

1. #include<iostream>
2. **using** **namespace** std;
3. **int** mul(**int**,**int**);
4. **float** mul(**float**,**int**);

7. **int** mul(**int** a,**int** b)
8. {
9. **return** a\*b;
10. }
11. **float** mul(**double** x, **int** y)
12. {
13. **return** x\*y;
14. }
15. **int** main()
16. {
17. **int** r1 = mul(6,7);
18. **float** r2 = mul(0.2,3);
19. std::cout << "r1 is : " <<r1<< std::endl;
20. std::cout <<"r2 is : "  <<r2<< std::endl;
21. **return** 0;
22. }

**Output:**

r1 is : 42

r2 is : 0.6

Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

**Causes of Function Overloading:**

* Type Conversion.
* Function with default arguments.
* Function with pass by reference.
* Type Conversion:

**Let's see a simple example.**

1. #include<iostream>
2. **using** **namespace** std;
3. **void** fun(**int**);
4. **void** fun(**float**);
5. **void** fun(**int** i)
6. {
7. std::cout << "Value of i is : " <<i<< std::endl;
8. }
9. **void** fun(**float** j)
10. {
11. std::cout << "Value of j is : " <<j<< std::endl;
12. }
13. **int** main()
14. {
15. fun(12);
16. fun(1.2);
17. **return** 0;
18. }

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

* Function with Default Arguments

**Let's see a simple example.**

1. #include<iostream>
2. **using** **namespace** std;
3. **void** fun(**int**);
4. **void** fun(**int**,**int**);
5. **void** fun(**int** i)
6. {
7. std::cout << "Value of i is : " <<i<< std::endl;
8. }
9. **void** fun(**int** a,**int** b=9)
10. {
11. std::cout << "Value of a is : " <<a<< std::endl;
12. std::cout << "Value of b is : " <<b<< std::endl;
13. }
14. **int** main()
15. {
16. fun(12);
18. **return** 0;
19. }

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

* Function with pass by reference

Let's see a simple example.

1. #include <iostream>
2. **using** **namespace** std;
3. **void** fun(**int**);
4. **void** fun(**int** &);
5. **int** main()
6. {
7. **int** a=10;
8. fun(a); // error, which f()?
9. **return** 0;
10. }
11. **void** fun(**int** x)
12. {
13. std::cout << "Value of x is : " <<x<< std::endl;
14. }
15. **void** fun(**int** &b)
16. {
17. std::cout << "Value of b is : " <<b<< std::endl;
18. }

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

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++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

1. #include <iostream>
2. **using** **namespace** std;
3. **class** Animal {
4. **public**:
5. **void** eat(){
6. cout<<"Eating...";
7. }
8. };
9. **class** Dog: **public** Animal
10. {
11. **public**:
12. **void** eat()
13. {
14. cout<<"Eating bread...";
15. }
16. };
17. **int** main(**void**) {
18. Dog d = Dog();
19. d.eat();
20. **return** 0;
21. }

Output:

Eating bread...

**How to overload operators in C++ programming?**

To overload an operator, a special operator function is defined inside the class as:

class className

{

... .. ...

public

returnType operator symbol (arguments)

{

... .. ...

}

... .. ...

};

* Here, returnType is the return type of the function.
* The returnType of the function is followed by operator keyword.
* Symbol is the operator symbol you want to overload. Like: +, <, -, ++
* You can pass arguments to the operator function in similar way as functions.

**Example: Operator overloading in C++ Programming**

#include <iostream>

using namespace std;

class Test

{

private:

int count;

public:

Test(): count(5){}

void operator ++()

{

count = count+1;

}

void Display() { cout<<"Count: "<<count; }

};

int main()

{

Test t;

// this calls "function void operator ++()" function

++t;

t.Display();

return 0;

}

**Output**

Count: 6

This function is called when ++ operator operates on the object of Test class (object t in this case).

In the program,void operator ++ () operator function is defined (inside Test class).

This function increments the value of count by 1 for t object.

### Things to remember

1. Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.).
2. Two operators = and & are already overloaded by default in C++. For example: To copy objects of same class, you can directly use = operator. You do not need to create an operator function.
3. Operator overloading cannot change the precedence and associatively of operators. However, if you want to change the order of evaluation, parenthesis should be used.
4. There are 4 operators that cannot be overloaded in C++. They are :: (scope resolution), . (member selection), .\* (member selection through pointer to function) and ?: (ternary operator).

### Following best practices while using operator overloading

Operator overloading allows you to define the way operator works (the way you want).

In the above example, ++ operator operates on object to increase the value of data member count by 1.

void operator ++()

{

count = count+1;

}

However, if you use the following code. It decreases the value of count by 100 when ++ operator is used.

void operator ++()

{

count = count-100;

}

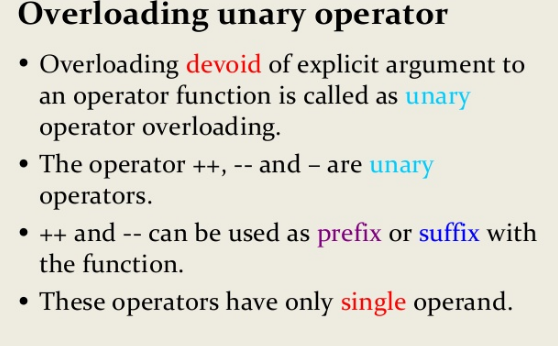
This may be technically correct. But, this code is confusing and, difficult to understand and debug.

It's your job as a programmer to use operator overloading properly and in consistent way.

In the above example, the value of count increases by 1 when ++ operator is used. However, this program is incomplete in sense that you cannot use code like:

t1 = ++t

It is because the return type of the operator function is void.



# Types of Operator Overloading in C++

**Operator Overloading:**

*C++ provides a special function to change the current functionality of some operators within its class which is often called as operator overloading. Operator Overloading is the method by which we can change the function of some specific operators to do some different task.*

This can be done by declaring the function, its syntax is,

Return\_Type classname :: operator op(Argument list)

{

Function Body

}

In the above syntax Return\_Type is value type to be returned to another object, operator op is the function where the operator is a keyword and op is the operator to be overloaded.

Operator function must be either non-static (member function) or friend function.

Operator Overloading can be done by using **three approaches**, they are

1. Overloading unary operator.
2. Overloading binary operator.
3. Overloading binary operator using a friend function.

Below are some criteria/rules to define the operator function:

* In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.
* In the case of a friend function, the binary operator should have only two argument and unary should have only one argument.
* All the class member object should be public if operator overloading is implemented.
* Operators that cannot be overloaded are **.** **.\*** **::** **?:**
* Operator cannot be used to overload when declaring that function as friend function **=** **()** **[]** **->**.

Refer this, for more rules of Operator Overloading

**Note**: The arguments in the operator overloading are passed only by reference, it will not work if arguments are passed by value, because a copy of the object is passed to operator (op)() function.

1. **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).

|  |
| --- |
| // C++ program to show unary operator overloading  #include <iostream>    using namespace std;    class Distance {  public:        // Member Object      int feet, inch;        // Constructor to initialize the object's value      Distance(int f, int i)      {          this->feet = f;          this->inch = i;      }        // Overloading(-) operator to perform decrement      // operation of Distance object      void operator-()      {          feet--;          inch--;          cout << "\nFeet & Inches(Decrement): " << feet << "'" << inch;      }  };    // Driver Code  int main()  {      // Declare and Initialize the constructor      Distance d1(8, 9);        // Use (-) unary operator by single operand      -d1;      return 0;  } |

**Output:**

Feet & Inches (Decrement): 7'8

In the above program, it shows that no argument is passed and no return\_type value is returned, because unary operator works on a single operand. (-) operator change the functionality to its member function.

**Note:** d2 = -d1 will not work, because operator-() does not return any value.

1. **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;      }        // Constructor to initialize the object's value      // Parametrized Constructor      Distance(int f, int i)      {          this->feet = f;          this->inch = i;      }        // Overloading (+) operator to perform addition of      // two distance object      Distance operator+(Distance& d2) // Call by reference      {          // Create an object to return          Distance d3;            // Perform addition of feet and inches          d3.feet = this->feet + d2.feet;          d3.inch = this->inch + d2.inch;            // Return the resulting object          return d3;      }  };    // Driver Code  int main()  {      // Declaring and Initializing first object      Distance d1(8, 9);        // Declaring and Initializing second object      Distance d2(10, 2);        // Declaring third object      Distance d3;        // Use overloaded operator      d3 = d1 + d2;        // Display the result      cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;      return 0;  } |

**Output:**

Total Feet & Inches: 18'11

Here in the above program,  
See Line no. 26, Distance operator+(Distance &d2), here return type of function is distance and it uses call by references to pass an argument.  
See Line no. 49, d3 = d1 + d2; here, d1 calls the operator function of its class object and takes d2 as a parameter, by which operator function return object and the result will reflect in the d3 object.

Pictorial View of working of Binary Operator:

1. **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.

|  |
| --- |
| // 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;      }        // Constructor to initialize the object's value      // Parametrized Constructor      Distance(int f, int i)      {          this->feet = f;          this->inch = i;      }        // Declaring friend function using friend keyword      friend Distance operator+(Distance&, Distance&);  };    // Implementing friend function with two parameters  Distance operator+(Distance& d1, Distance& d2) // Call by reference  {      // Create an object to return      Distance d3;        // Perform addition of feet and inches      d3.feet = d1.feet + d2.feet;      d3.inch = d1.inch + d2.inch;        // Return the resulting object      return d3;  }    // Driver Code  int main()  {      // Declaring and Initializing first object      Distance d1(8, 9);        // Declaring and Initializing second object      Distance d2(10, 2);        // Declaring third object      Distance d3;        // Use overloaded operator      d3 = d1 + d2;        // Display the result      cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;      return 0;  } |

**Output:**

Total Feet & Inches: 18'11

Here in the above program, operator function is implemented outside of class scope by declaring that function as the friend function.

In these ways, an operator can be overloaded to perform certain tasks by changing the functionality of operators.

**Manipulating of strings in C++ by operator overloading**

Manipulating of strings in C++ by operator overloading using character arrays, pointers and string functions. There are no operators for manipulating the strings. There are no direct operator that could act upon the strings or manipulate the strings.

Although there are these limitations exist in C++, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal number. We can manipulate strings by operator overloading as this is not achieved by operators only.  
  
**For example :**  
We should be able to use statement like this in manipulating strings using operator overloading -

string3 = string1 + string2;

Let us see an simple example of above statement and use to manipulate the strings :

#include<iostream>

using namespace std;

int main ()

{

string First = "This is First String and ";

string Second = "This is Second String.";

string Third = First + Second;

cout << Third;

return 0;

}

String Manipulation in C++ contains many string functions which we can use to manipulate the strings. In some compilers we can use them only by including iostream but in compilers which gives error have to include string library(#include<string>). Those functions can be Stringname.length , Stringname[expression](this expression is a number of character to show from the string), and many more.  
  
Let us see different functions can be used to manipulate the strings by this program example in C++ :

#include<iostream>

#include<string>

using namespace std;

int main ()

{

string Example = "Techoschool !";

cout << Example.length() << "\n\n";

// This length manipulator tells the length of string entered

cout << Example[5] << "\n\n";

// This manipulator takes the number in parameter to show the character on that value

cout << Example.substr(0, 5) << "\n\n";

// This substr manipulator prints the character upto 5 character in string

cout << Example.erase(0, 5) << "\n\n";

// This erase the number of characters mentioned from string and print rest string

cout << Example.insert(0, "Hello ") << "\n\n";

// This inserts the string to exist string on the number of place of string mentioned

cout << Example.append(5, '\*') << "\n\n";

// This appends the given string to the existing string from the last position

cout << Example.replace(0, 5, "Replaced Character");

// This replace string according to position entered with the string mentioned

return 0;

}

**RULES FOR OVERLOADING OPERATOR :-**  
  
1) Operators already predefined in the C++ compiler can be only overloaded.  
  
2) Overloading cannot change operator's templates.for eg: the increment operator ++ is used only as unary operator, it cannot be used as binary operator.  
  
3) Overloading an operator does not change its basic meaning. for eg: the ++ operator cannot be overloaded to subtract two objects.  
  
4) Unary operator, overloaded by means of a member of a member function, takes no explicit argument and return no explicit values.But those overloaded by means of a friend function take one reference argument (the object of the relevent class).  
  
5) Binary operators overloaded through a member function take one explicit argument and those overloaded through a friend function take two explicit arguments.