Function Overloading in C++

If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

Different ways to Overload a Function

1. By changing number of Arguments.
2. By having different types of argument

Function Overloading: Different Number of Arguments

In this type of function overloading we define two functions with same names but different number of parameters of the same type. For example, in the below mentioned program we have made two sum() functions to return sum of two and three integers.

```
// first definition
int sum (int x, int y)
{
   cout << x+y;
}

// second overloaded defintion  int sum(int x, int y, int z)
{
   cout << x+y+z;
}
```

Here sum() function is said to overloaded, as it has two defintion, one which accepts two arguments and another which accepts three arguments. Which sum() function will be called, depends on the number of arguments.

Example 1

```
int main()
{
   // sum() with 2 parameter will be called  sum (10, 20);
```

```
    //sum() with 3 parameter will be called
    sum(10, 20, 30);
}
```

2. Function Overloading: Different Data type of Arguments

In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different. For example in this program, we have two sum() function, first one gets two integer arguments and second one gets two double arguments.

Example 2

```
#Include<iostream.h>
#include<conio.h>

// first definition  int sum(int x, int y)
{
    cout<< x+y;
}

// second overloaded defintion  double sum(double x, double y)
{
    cout << x+y;
}
int main()
{
    sum (10,20);  sum(10.5,20.5);
}
```

Example 3 : Addition

```
#include <iostream>  using namespace std;
/* Function arguments are of different data type */
int add(int, int);
float add(float, float);  int main()
{
```

```
 int a, b, x;  float c, d, y;
 cout << "Enter two integers\n";  cin >> a >> b;
 x = add(a, b);
 cout << "Sum of integers: " << x << endl;  cout << "Enter two floating point numbers\n";
 cin >> c >> d;
 y = add(c, d);
 cout << "Sum of floats: " << y << endl;  return 0;
}
int add(int x, int y)
{
 int sum;  sum = x + y;  return sum;
}
float add(float x, float y)
{
 float sum;  sum = x + y;  return sum;
}
```

Explanation:
We created two functions "add" for integer and float data types, you can create any number of them of the same name as required. Just make sure a compiler will be able to determine which one to call. In the program, you can create add function for long, double, and other data types.

Example 4:

```
#include <iostream>
using namespace std;  class Addition
{
public:
   int sum(int num1,int num2)
{
       return num1+num2;
 }
   int sum(int num1,int num2, int num3)
 {
      return num1+num2+num3;
 }
};
int main(void)
{
   Addition obj;  cout<<obj.sum(20, 15)<<endl;
```

```
   cout<<obj.sum(81, 100, 10);
   return 0;
}
```

Output:

```
35
191
```

OPERATING OVERLOADING

The subscript operator [] is normally used to access array elements. This operator can be overloaded to enhance the existing functionality of C++ arrays.

Following example 5 explains how a subscript operator [] can be overloaded.

Syntax of Operator Overloading

```
returntype classname :: operator operator_to_overload ([parameter])

{

   statement(s);

   ... ... ...

}
```

Example 5:

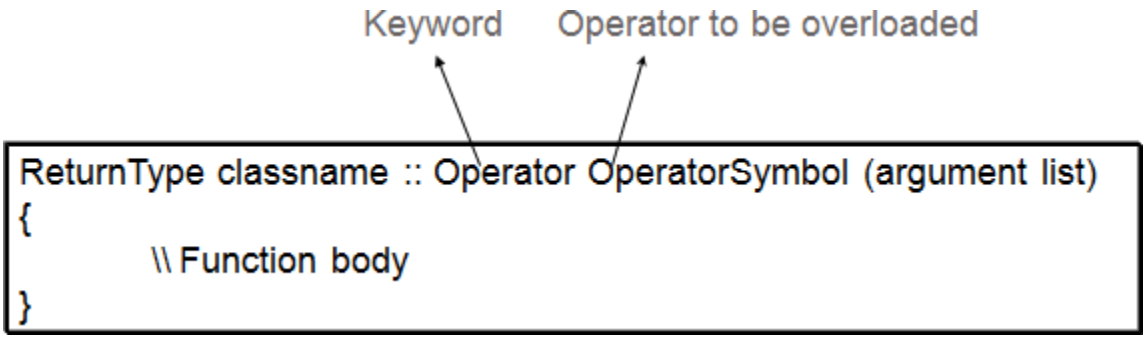
```
#include <iostream.h>  using
namespace std;  const int SIZE
= 10;  class safearay {
  private:
     int arr[SIZE];  public:
     safearay() {  register int i;
        for(i = 0; i < SIZE; i++) {
          arr[i] = i;
       }
    }
    int &operator[](int i) {
```

```
        if( i > SIZE ) {
          cout << "Index out of bounds" <<endl;
          // return first element.  return arr[0];
        }
        return arr[i];
      }
};
int main() {  safearay A;
cout << "Value of A[2] : " << A[2] <<endl;
cout << "Value of A[5] : " << A[5]<<endl;  cout
  << "Value of A[12] : " << A[12]<<endl;
  return 0;
}
```

Output

```
Value of A[2] : 2  Value of
A[5] : 5  Index out of bounds
Value of A[12] : 0
```

Operator Overloading Syntax



Syntax of Operator Overloading

```
returntype classname :: operator operator_to_overload ([parameter])
{
   statement(s);
   ... ... ...
}
```

Exampl6

```
#include <iostream>  #include <conio.h>  using namespace std;

class example
{
   int a,b;  public:
      void input()
      {
         cout<<"Enter a and b: ";  cin>>a>>b;
      }
      void operator -() //operator function as a member function
      {
         a=-a;
         b=-b;
      }
      void display()
      {
         cout<<"a="<<a<<endl<<"b="<<b<<endl;
      }
};

int main()
{
   example e;  e.input();
   cout<<"Before overloading unary minus operator"<<endl;
   e.display();
   -e;
   cout<<"After overloading unary minus operator"<<endl;
   e.display();
   getch();  return 0;
}
```

Rules for operator overloading

1. Only existing member can be overloaded. We can't create our own operator to overload.
2. The overloaded operator must have at least one operand of user-defined type.
3. Overloaded operators follow the syntax rules of original operators. This means we can't change the basic meaning of an operator.

4. Some operators can't be overloaded. They are: member access operator (.), pointer to member access operator (.*), scope resolution operator (::), size operator (sizeof), ternary operator (? :).
5. We can't use friend function to overload some operators. They are: assignment operator (=), function call operator (()), subscripting operator ([]), class member access operator (->).
6. If the operator function is a friend function then it will have one argument for unary operator and two argument for binary operator. If the operator function is a non static member function then it will have no arguments for unary operators and one argument for binary operators.
7. When binary operators are overloaded through member function, the left hand operand must be an object of the relevant class