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,

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 functions 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. }
```

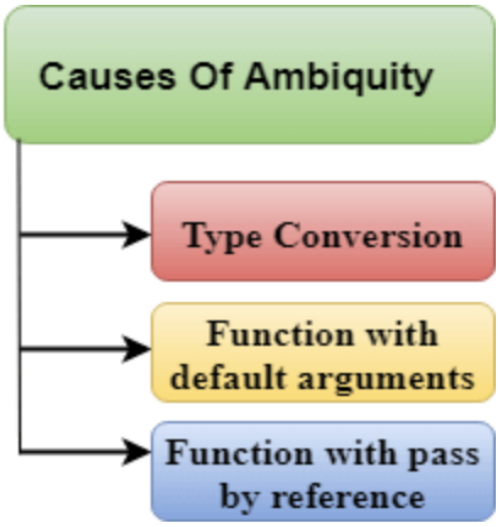
Output:

30
55

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.




```
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.

C++ Operator Overloading

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as **operator overloading**. For example,

Suppose we have created three objects c1, c2 and result from a class named Complex that represents complex numbers.

Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers of c1 and c2 by writing the following code:

result = c1 + c2;

Note: *We cannot use operator overloading for fundamental data types like int, float, char and so on.*

Syntax for C++ Operator Overloading

To overload an operator, we use a special operator function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

```
class className {
   ... .. ...
   public
     returnType operator symbol (arguments) {
        ... .. ...
     }
   ... .. ...
};
```

Here,

- returnType is the return type of the function.
- operator is a keyword.
- symbol is the operator we want to overload. Like: +, <, -, ++, etc.
- arguments is the arguments passed to the function.

Operator that cannot be overloaded are as follows:

There are 4 operators that cannot be overloaded in C++. They are:

1. :: (scope resolution)

2. . (member selection)
3. .* (member selection through pointer to function)
4. ?: (ternary operator)

Operator Overloading in Unary Operators

Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

Example1: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix

#include <iostream>
using namespace std;

class Count {
  private:
        int value;

  public:

        // Constructor to initialize count to 5
        Count() {
        Value = 5;
        }

        // Overload ++ when used as prefix
        void operator ++ () {
        ++value;
        }

        void display() {
        cout << "Count: " << value << endl;
        }
};

int main() {
        Count count1;

        // Call the "void operator ++ ()" function
        ++count1;

        count1.display();
```

```
    return 0;
}
```

Output

Count: 6

Here, when we use ++count1;, the void operator ++ () is called. This increases the value attribute for the object count1 by 1.

Note: *When we overload operators, we can use it to work in any way we like. For example, we could have used ++ to increase value by 100.*

However, this makes our code confusing and difficult to understand. It's our job as a programmer to use operator overloading properly and in a consistent and intuitive way.

The above example works only when ++ is used as a prefix. To make ++ work as a postfix we use this syntax.

```
void operator ++ (int) {

   // code

}
```

Notice the int inside the parentheses. It's the syntax used for using unary operators as postfix; it's not a function parameter.

Example 2: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix and postfix

#include <iostream>
using namespace std;

class Count {
  private:
        int value;

  public:
```

```
        // Constructor to initialize count to 5
        Count() : value(5) {}


        // Overload ++ when used as prefix
        void operator ++ () {
        ++value;
        }


        // Overload ++ when used as postfix
        void operator ++ (int) {
        value++;
        }

        void display() {
        cout << "Count: " << value << endl;
        }
};

int main() {
        Count count1;

        // Call the "void operator ++ (int)" function
        count1++;
        count1.display();

        // Call the "void operator ++ ()" function
        ++count1;

        count1.display();
        return 0;
}
```

Output

Count: 6

Count: 7

Example 3: Return Value from Operator Function (++ Operator)

```
#include <iostream>
using namespace std;

class Count {
  private:
        int value;

  public
        :
        // Constructor to initialize count to 5
        Count() : value(5) {}

        // Overload ++ when used as prefix
        Count operator ++ () {
        Count temp;

        // Here, value is the value attribute of the calling object
        temp.value = ++value;

        return temp;
        }

        // Overload ++ when used as postfix
        Count operator ++ (int) {
        Count temp;

        // Here, value is the value attribute of the calling object
        temp.value = value++;

        return temp;
        }

        void display() {
        cout << "Count: " << value << endl;
        }
};

int main() {
        Count count1, result;

        // Call the "Count operator ++ ()" function
        result = ++count1;
        result.display();
```

```
        // Call the "Count operator ++ (int)" function
        result = count1++;
        result.display();

        return 0;
}
```

Output

```
Count: 6
Count: 6
```

Here, we have used the following code for prefix operator overloading:

```
// Overload ++ when used as prefix
Count operator ++ () {
   Count temp;

   // Here, value is the value attribute of the calling object
   temp.value = ++value;

   return temp;
}
```

The code for the postfix operator overloading is also similar. Notice that we have created an object temp and returned its value to the operator function.

Also, notice the code

```
temp.value = ++value;
```

The variable value belongs to the count1 object in main() because count1 is calling the function, while temp.value belongs to the temp object.

Operator Overloading in Binary Operators

Binary operators work on two operands. For example,

```
result = num + 9;
```

Here, + is a binary operator that works on the operands num and 9.

When we overload the binary operator for user-defined types by using the code:

```
obj3 = obj1 + obj2;
```

The operator function is called using the obj1 object and obj2 is passed as an argument to the function.

Example 4: C++ Binary Operator Overloading

```
// C++ program to overload the binary operator +
// This program adds two complex numbers

#include <iostream>
using namespace std;

class Complex {
  private:
        float real;
        float imag;

  public:
        // Constructor to initialize real and imag to 0
        Complex() : real(0), imag(0) {}

        void input() {
        cout << "Enter real and imaginary parts respectively: ";
        cin >> real;
        cin >> imag;
        }

        // Overload the + operator
        Complex operator + (const Complex& obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
        }

        void output() {
        if (imag < 0)
        cout << "Output Complex number: " << real << imag << "i";
        else
        cout << "Output Complex number: " << real << "+" << imag << "i";
        }
};

int main() {
        Complex complex1, complex2, result;
```

```
        cout << "Enter first complex number:\n";
        complex1.input();

        cout << "Enter second complex number:\n";
        complex2.input();

    // complex1 calls the operator function
    // complex2 is passed as an argument to the function
        result = complex1 + complex2;
        result.output();

        return 0;
}
```

Output

```
Enter first complex number:
Enter real and imaginary parts respectively: 9 5
Enter second complex number:
Enter real and imaginary parts respectively: 7 6
Output Complex number: 16+11i
```

In this program, the operator function is:

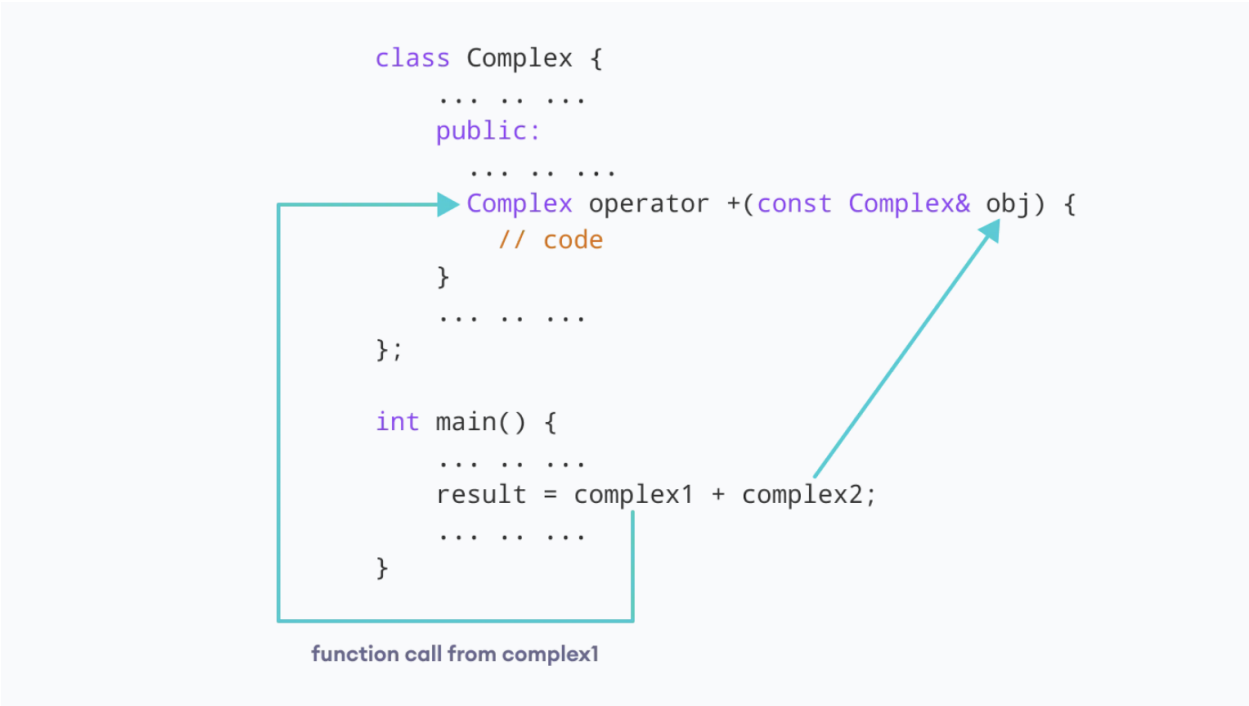
```
Complex operator + (const Complex& obj) {
    // code
}
```

Instead of this, we also could have written this function like:

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

However,

- using & makes our code efficient by referencing the complex2 object instead of making a duplicate object inside the operator function.
- using const is considered a good practice because it prevents the operator function from modifying complex2.



Things to Remember in C++ Operator Overloading

1. Two operators = and & are already overloaded by default in C++. For example, to copy objects of the same class, we can directly use the = operator. We do not need to create an operator function.
2. Operator overloading cannot change the precedence and associativity of operators. However, if we want to change the order of evaluation, parentheses should be used.