Operator Overloading in C++

C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Operator overloading in C++ is a powerful feature that allows you to redefine the behavior of operators for user-defined data types.

What are Operators?

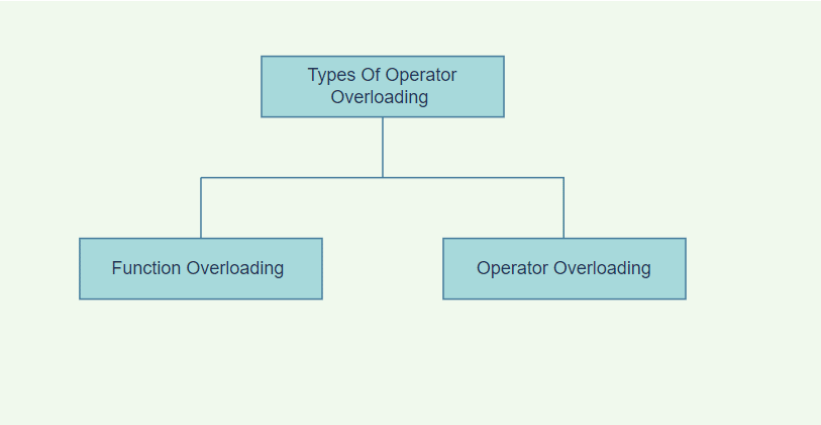
Operators in programming languages are symbols or keywords that perform specific operations on operands to produce results.

These operations can include arithmetic operations like addition and subtraction, comparison operations like equal to and not equal to, logical operations like AND and OR, and more. In essence, operators enable you to manipulate data and control the flow of your code.

What are the types of operator overloading?

There are two types of operator overloading:

- Function overloading.
- Operator overloading.



Function Overloading in C++

The process of having two or more functions with the same name but with different parameters (arguments) is called function overloading. The function is redefined by either using different types of arguments or a different number of arguments. It is only through these differences that a compiler can differentiate between functions.

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types –

```
using namespace std;

class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
```

When the above code is compiled and executed, it produces the following result –

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Operators Overloading in C++

Operator overloading in C++ program can add special features to the functionality and behavior of already existing operators, such as arithematics and other operations.

The mechanism of giving special meaning to an operator is known as operator overloading. For example, we can overload an operator '+' in a class-like string to concatenate two strings by just using +.

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Operations that can be performed:

- Arithmetic operations: + – * / %
- Logical operations: && and ||
- Relational operations: == != >= <=
- Pointer operators: & and *
- Memory management operator: new, delete []

```
Box operator+(const Box&);
```

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows –

```
Box operator+(const Box&, const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below –

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

class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
      void setLength( double len ) {
         length = len;
      }
      void setBreadth( double bre ) {
         breadth = bre;
      }
      void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      Box operator+(const Box& b) {
         Box box; //creates new object box that hold the result of
adding two boxes object
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }

   private:
      double length;       // Length of a box
      double breadth;      // Breadth of a box
      double height;       // Height of a box
};

// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
```

```
   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;

}
```

When the above code is compiled and executed, it produces the following result –

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Lab Tasks

Task 1:

Write a C++ program to define a Rectangle class that represents a rectangle with two properties: length and width. Implement the following features:

- Implement two overloaded functions named area():

- One version takes no parameters and calculates the area of a rectangle using the object's length and width.
- Another version takes two parameters (length and width) and calculates the area of a rectangle with these values.
- Overload the == operator to compare two Rectangle objects based on their area. If the areas of two rectangles are the same, the operator should return true; otherwise, it should return false.

Task 2:

Write a C++ program to overload the `*` operator for a `complex` class. The `Complex` class should represent complex numbers and contain two private data members: `real` and `imaginary`. Overload the `*` operator to multiply two complex numbers and return the resulting `Complex` object.

- **Hint**: Recall that if (a+bi) and (c+di) are two complex numbers, their product is (ac−bd)+(ad+bc)i.

Task 3:

Write a C++ program to define a class **Vector3D** that represents a 3D mathematical vector with three properties: **x**, **y**, and **z**.

Implement the following features:

1. **Function Overloading:**
    - Overload a function named magnitude()
        - One version that takes **no parameters** and calculates the magnitude of the current object using the formula:

        - Another version that takes **three parameters (x, y, z)** and returns the magnitude for those values without changing the object's data members.
2. **Operator Overloading:**

- Overload the + operator to **add two Vector3D objects** and return the resulting vector.
- Overload the == operator to **compare two Vector3D objects** — return true if their magnitudes are equal, otherwise false.

3. **In main() Function:**
   - Create two Vector3D objects.
   - Display their magnitudes using both versions of the magnitude() function.
   - Add the two vectors using the overloaded + operator and display the result.
   - Compare the two vectors using the overloaded == operator.

Lab #07 Marks distribution

	ER1	ER6	ER8
Task	2 points	2 points	4 points

Lab # 07 Rubric Evaluation Guideline:

#	Qualities & Criteria	0 < Poor <=0.5	0.5 < Satisfactory <= 1	1 < Excellent <=1.5
ER 1	**Task Completion**	Minimal or no program functionality was achieved.	Some tasks were completed, but the program has errors or incomplete functionalities.	All tasks were completed, and the program runs without errors.
#	**Qualities & Criteria**	**0 < Poor <=0.5**	**0.5 < Satisfactory <= 1**	**1 < Excellent <=1.5**
ER 6	Program Output	Output is inaccurate or poorly presented.	Output is mostly accurate but may lack labels, captions, or formatting.	Output is clear, accurate, and well presented with labels, captions, and proper formatting.
#	**Qualities & Criteria**	**0 < Poor <= 0.5**	**0.5< Satisfactory <= 1**	**1 < Excellent <= 2**
ER 8	Question & Answer	Answers some questions but not confidently or based on lab task knowledge.	Answers most questions confidently and based on lab task knowledge.	Answers all questions confidently and demonstrates a deep understanding of the given lab task.