Polymorphism: Function and Operator Overloading in C++

Polymorphism is one of the key principles of Object-Oriented Programming (OOP) and plays a significant role in C++ programming. The term polymorphism means "many forms" and refers to the ability of functions, methods, or operators to behave differently based on their context. In C++, polymorphism is primarily implemented in two forms: **compile-time polymorphism** and **run-time polymorphism**. This note focuses on compile-time polymorphism, which includes **function overloading** and **operator overloading**.

Function Overloading

Function overloading is a feature in C++ that allows multiple functions to have the same name but differ in terms of the number or type of their arguments. It enhances code readability and reduces the need for complex naming conventions. Function overloading is resolved at compile time, and the compiler decides which function to invoke based on the arguments passed to it.

Characteristics of Function Overloading:

1. **Same Name, Different Signatures**: Functions must have the same name but a different number or type of parameters to be overloaded.
2. **Return Type Irrelevance**: The return type of a function does not play a role in function overloading. For example, defining two functions that differ only by their return type will result in a compilation error.
3. **Scope-Specific Overloading**: Function overloading is applicable only to functions in the same scope.

Syntax Example:

```
#include <iostream>

using namespace std;


// Function to calculate the area of a rectangle

int area(int length, int width) {

   return length * width;

}


// Overloaded function to calculate the area of a square

int area(int side) {

   return side * side;

}
```

```
// Overloaded function to calculate the area of a circle
double area(double radius) {
    return 3.14159 * radius * radius;
}

int main() {
    cout << "Area of rectangle: " << area(5, 10) << endl;
    cout << "Area of square: " << area(7) << endl;
    cout << "Area of circle: " << area(5.5) << endl;
    return 0;
}
```

In this example, the area() function is overloaded three times to handle different shapes (rectangle, square, and circle). Based on the arguments provided, the compiler determines which version of the function to execute.

Operator Overloading

Operator overloading allows built-in operators in C++ (such as +, -, *, =) to be redefined for user-defined types. This makes custom data types (e.g., classes) behave like built-in types when used with these operators. Operator overloading is achieved using special functions called **operator functions**.

Characteristics of Operator Overloading:

1. **Customization for User-Defined Types**: Operators can be redefined to perform specific operations on objects of user-defined classes.
2. **Syntax and Behavior Consistency**: The syntax of the overloaded operator remains the same as the built-in operators, maintaining code readability.
3. **Operator Restrictions**:
    - Not all operators can be overloaded (e.g., ::, .*, sizeof).
    - Overloading operators does not change their precedence or associativity.
4. **Friend Function Option**: Some operators (like << and >>) can be overloaded using friend functions to allow non-member access to private or protected members.

Syntax Example:

```
#include <iostream>
```

```
using namespace std;

class Complex {
private:
    double real;
    double imag;

public:
    // Constructor
    Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}

    // Overload '+' operator
    Complex operator+(const Complex& obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }

    // Overload '<<' operator for displaying
    friend ostream& operator<<(ostream& out, const Complex& obj) {
        out << obj.real << " + " << obj.imag << "i";
        return out;
    }
};

int main() {
    Complex c1(3.5, 2.5), c2(1.5, 3.0);

    // Using overloaded '+' operator
    Complex c3 = c1 + c2;

    // Using overloaded '<<' operator
```

```
    cout << "Result: " << c3 << endl;

    return 0;
}
```

In this example, the + operator is overloaded to perform addition on complex numbers, and the << operator is overloaded to display the result.

Advantages of Function and Operator Overloading

1. **Enhanced Readability**: Function and operator overloading eliminate the need for separate function names for similar operations, resulting in cleaner and more intuitive code.
2. **Improved Code Reusability**: Reusing the same function or operator name for different purposes simplifies code maintenance.
3. **Better Integration with OOP Principles**: Operator overloading enhances encapsulation by allowing user-defined types to integrate seamlessly with existing operators.