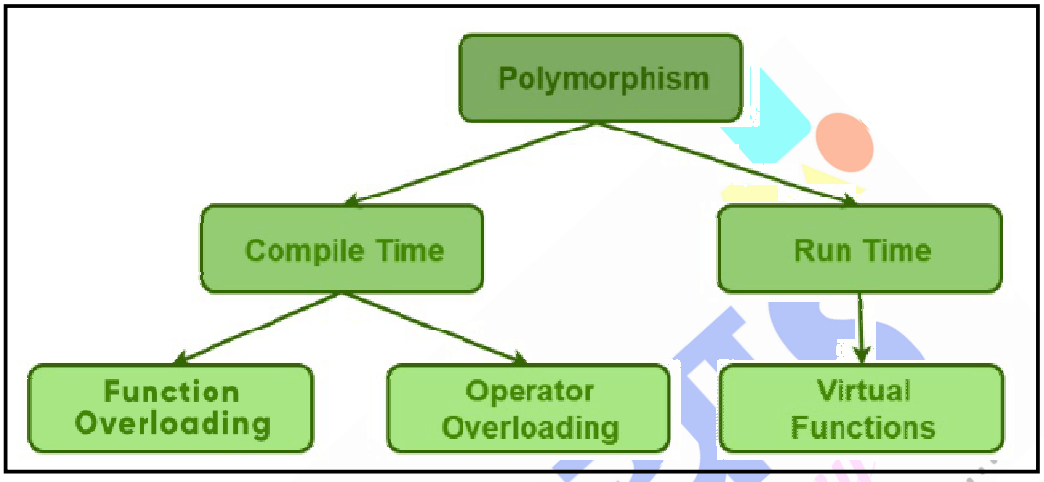
Polymorphism in C Plus Plus

Module 3 - Operator Overloading in C Plus Plus

The word "polymorphism" comes from the Greek words *poly* (many) and *morph* (forms), meaning "many forms."



Overloading refers to the use of the same thing for different purposes.

Operator overloading allows you to redefine the functionality of an operator for user-defined data types. This helps to perform operations on objects as naturally as you would on built-in types.

We can overload all the C++ operators except the following:
- Class members access operator (. , .*)
- Scope resolution operator (: :)
- Size operator(sizeof)
- Condition operator (? :)

Although the semantics of an operator can be extended, we can't change its syntax, the grammatical rules that govern its use such as the no of operands precedence and associativity. For example the **multiplication operator** will enjoy higher precedence than the **addition operator.**

When an operator is overloaded, its original meaning is not lost. For example, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

Introduction to 'C++' Language

Syntax Rules:
- At least one operand must be a user-defined type (class or struct).
- You cannot change the precedence or associativity of operators.
- Overloading must be explicitly defined for each operator.

Defining Operator Overloading:
To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied . This is done with the help of a special function called operator function, which describes the task.

Syntax:-
```
return-type class-name :: operator op( arg-list)
{
function body
}
```

Where return type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator, **operator op is the function name.**

Types of Operator Overloading
Operator overloading can be categorized into two main types based on the number of operands involved:
1. **Unary Operator Overloading**
2. **Binary Operator Overloading**

Unary Operator Overloading

A **unary operator** works with only one operand. Examples include -, ++, --, and !.
Key Points
- The operator function takes no arguments when overloaded as a member function.
- It takes one argument (the object itself) when overloaded as a non-member function or a friend function.

Solved Example: Overloading - (Negation) Operator

Here's an example of overloading the unary - operator for a class Number:

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

class Number {
private:
    int value;
```

```
public:
    // Constructor
    Number(int v = 0) : value(v) {}

    // Overloading the unary - operator
    Number operator-() {
        return Number(-value);
    }

    // Display function
    void display() const {
        cout << "Value: " << value << endl;
    }
};

int main() {
    Number n1(10);

    Number n2 = -n1; // Calls the overloaded unary - operator

    cout << "Original: ";
    n1.display();

    cout << "Negated: ";
    n2.display();

    return 0;
}
```

Output of the Given Code

Original: Value: 10
Negated: Value: -10

Explanation

1. The Number class has a private member variable value and a unary - operator overloaded in the operator- function.
2. When Number n2 = -n1; is executed:
   - The operator-() function is invoked for the n1 object.
   - This creates a new Number object with the value -10 (the negation of n1.value).
3. The display() function is called for both n1 and n2 to print their values:
   - n1.display() outputs Value: 10.
   - n2.display() outputs Value: -10.

Binary Operator Overloading

A **binary operator** works with two operands. Examples include +, -, *, /, %, and ==.

Key Points

• The operator function takes one argument (the second operand) when overloaded as a member function.

• It takes two arguments when overloaded as a non-member or friend function.

Solved Example: Overloading + Operator

Here's an example of overloading the binary + operator for a class Complex:

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

class Complex {
private:
    float real, imag;

public:
    // Constructor
    Complex(float r = 0, float i = 0) : real(r), imag(i) {}

    // Overloading the binary + operator
    Complex operator+(const Complex& c) {
        return Complex(real + c.real, imag + c.imag);
    }

    // Display function
    void display() const {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3.5, 2.5), c2(1.5, 4.5);

    Complex c3 = c1 + c2; // Calls the overloaded binary + operator

    cout << "Result: ";
    c3.display();

    return 0;
}
```

Output of the Given Code
Result: 5 + 7i

Explanation
1. **Class Definition:**
   - The Complex class has private member variables real and imag, which represent the real and imaginary parts of a complex number.
   - The constructor Complex(float r = 0, float i = 0) initializes these variables.
2. **Overloaded + Operator:**
   - The operator+ function adds the corresponding real and imag parts of two Complex objects.
   - It returns a new Complex object with the computed values.
3. **Execution in main:**
   - Two Complex objects, c1 (3.5 + 2.5i) and c2 (1.5 + 4.5i), are created.
   - The statement **Complex c3 = c1 + c2;** invokes the overloaded + operator.
     - real part: 3.5 + 1.5 = 5.0
     - imag part: 2.5 + 4.5 = 7.0
     - The result is a new Complex object, c3, with real = 5.0 and imag = 7.0.
4. **Output:**
   - The display() function is called on c3, which outputs 5 + 7i.