🎮

Operator Overloading

What is Operator Overloading?

Operator overloading in C++ allows us to redefine the behavior of operators (like `+`, `-`, `*`, `/`, `==`, etc.) for user-defined types (such as classes). It enables objects of a class to be manipulated similarly to built-in data types.

Why Use Operator Overloading?

1. **Improves Code Readability**: Makes user-defined types behave naturally.
2. **Enhances Code Maintainability**: Reduces unnecessary function calls.
3. **Supports Object-Oriented Programming**: Allows operator customization for objects.

Types of Operator Overloading

1. **Unary Operator Overloading** (e.g., `++`, `-`, , `!`)
2. **Binary Operator Overloading** (e.g., `+`, , , `/`, `==`, `!=`, `<`, `>`)
3. **Overloading I/O Operators (`<<` and `>>`)**
4. **Overloading Assignment Operator (`=`)**
5. **Overloading Function Call Operator (`()`)**
6. **Overloading Subscript Operator (`[]`)**
7. **Overloading Pointer Operators (`>`,)**

How to Overload Operators?

- Operator functions must be **non-static member functions** or **friend functions**.
- Syntax:

```
return_type operator symbol (argument_list)
```

Unary Operator Overloading (Example: Operator)

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

class Number {
    int value;

public:
    Number(int v) : value(v) {}

    // Overloading Unary '-' Operator
    Number operator-() {
        return Number(-value);
    }

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

int main() {
    Number num(10);
    Number negNum = -num;  // Using overloaded unary '-'

    num.display();     // Value: 10
    negNum.display();  // Value: -10
    return 0;
}
```

Binary Operator Overloading (Example: `+` Operator)

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

class Complex {
    int real, imag;

public:
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}

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

    void display() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2;  // Using overloaded '+'

    c3.display();  // 4 + 6i
    return 0;
}
```

Overloading I/O Operators (`<<` and `>>`)

The `<<` and `>>` operators must be **friend functions** since the left operand (`cout` or `cin`) is not an object of the class.

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

class Complex {
    int real, imag;

public:
    Complex(int r = 0, int i = 0) : real(r), imag(i) {}

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

    // Overloading '>>' for input
    friend istream& operator>>(istream& in, Complex& obj) {
        cout << "Enter real and imaginary parts: ";
        in >> obj.real >> obj.imag;
        return in;
    }
};

int main() {
    Complex c;
    cin >> c;  // Input: 5 6
    cout << "Complex number: " << c << endl;  // Output: 5 + 6i
    return 0;
}
```

Overloading Assignment Operator (`=`)

When overloading `=`, we need to handle **deep copying** to prevent memory leaks in dynamic objects.

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

class String {
    char* str;

public:
    String(const char* s = "") {
        str = new char[strlen(s) + 1];
        strcpy(str, s);
    }

    // Overloading Assignment Operator
    String& operator=(const String& obj) {
        if (this != &obj) {
            delete[] str;
            str = new char[strlen(obj.str) + 1];
            strcpy(str, obj.str);
        }
        return *this;
    }

    void display() { cout << str << endl; }

    ~String() { delete[] str; }
};

int main() {
    String s1("Hello");
    String s2;
    s2 = s1;  // Using overloaded '='
    s2.display();  // Output: Hello
    return 0;
}
```

Overloading Subscript Operator (`[]`)

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

class Array {
    int arr[5];

public:
    Array() { for (int i = 0; i < 5; i++) arr[i] = i * 10; }

    // Overloading '[]' Operator
    int operator[](int index) {
        if (index < 0 || index >= 5) {
            cout << "Index out of bounds!\n";
            return -1;
        }
        return arr[index];
    }
};

int main() {
    Array a;
    cout << "Element at index 2: " << a[2] << endl;  // Output: 20
    return 0;
}
```

Overloading Function Call Operator (`()`)

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

class Functor {
public:
    void operator()(string text) {
        cout << "Calling functor: " << text << endl;
    }
};

int main() {
    Functor obj;
    obj("Hello, World!");  // Calls overloaded '()'
    return 0;
}
```

Operators That Cannot Be Overloaded

Some operators cannot be overloaded in C++:

1. `::` (Scope resolution)
2. `.*` (Pointer-to-member)
3. `.` (Member access)

4. `sizeof` (Size determination)
5. `typeid` (Run-time type identification)

Conclusion

- Operator overloading allows user-defined types to behave like built-in types.
- Unary and binary operators can be overloaded as **member** or **friend** functions.
- Some operators (like `<<`, `>>`, `=`) require special handling.
- Not all operators can be overloaded.