Operator Overloading in C++

1. Introduction

Today we'll explore one of C++'s most powerful and expressive features: operator overloading.

Operator overloading allows you to define custom behavior for operators (like +, -, <<, ==, etc.) when applied to user-defined types (e.g., classes or structs).

💡 Goal: Make your objects behave like built-in types—intuitive, readable, and natural.

Example:

```
Vector a(1, 2), b(3, 4);
Vector c = a + b;          // ← This is possible with
                           // operator overloading!
std::cout << c;            // ← And this too!
```

2. Why Overload Operators?

- Improve readability: a + b is clearer than a.add(b)
- Enable natural syntax: Especially for mathematical or collection-like types
- Integrate with standard libraries: e.g., use your class with std::sort by overloading <
- Maintain consistency: Objects act like built-in types

⚠ Rule of thumb: Overload operators only when the meaning is obvious and consistent with built-in types.

3. How to Overload Operators

You can overload operators as:

- Member functions (inside the class)
- Non-member (free) functions (often as friend functions)

General Syntax:

```
// As member function
ReturnType operator@(Parameters);
// As non-member function
ReturnType operator@(const Type& lhs, const Type&
rhs);
```

@ represents the operator symbol (e.g., +, ==, <<)

4. Member vs. Non-Member Overloading

ASPECT	MEMBER FUNCTION	NON-MEMBER FUNCTION
Implicit `this`	Yes (left operand to `this`)	No (both operands passed explicitly)
Access to private members	Yes	Only if declared `friend`
Symmetry	Non symmetric (left side must be your type)	Symmetric (e.g., int + MyType possible)
Common for	Assignment (=), subscript(`[]`), function call (`()`)	Stream insertion (<<), arithmetic (+, −), comparison (==, <)

4. Member vs. Non-Member Overloading

Best practice:

- Use member functions for operators that modify the left operand (e.g., +=, =)
- Use non-member functions for symmetric operators (e.g., +, ==)

Overloading Arithmetic Operators in C++

1. Introduction

Arithmetic operators (+, -, *, /, %, +=, -= etc.) are essential for mathematical and logical operations. In C++, you can customize their behavior for your own classes—enabling natural, expressive syntax like:

```
Vector a(1, 2), b(3, 4);
Vector c = a + b;       // Binary +
Vector d = -a;          // Unary -
a += b;                 // Compound assignment
```

we'll explore:

- Unary vs. binary arithmetic operators
- How to overload them as member functions and non-member (free) functions
- Best practices for consistency, efficiency, and safety

2. Unary vs. Binary Arithmetic Operators

TYPE	OPERATORS	OPERANDS
Unary	+,-,++,--	One operand (e.g., -x, ++x)
Binary	+, -, *, /, %, +=	Two operands (e.g., a + b)

Note: + and - can be both unary and binary (context determines which is used).

3. General Guidelines for Overloading

✅ Prefer non-member functions for symmetric binary operators

- Example: a + b should work the same as b + a
- If a is your type and b is int, a member function won't allow 5 + a

✅ Prefer member functions for compound assignment (+=, -=)

- They modify the left-hand object → natural as members

✅ Implement binary operators in terms of compound assignment

- Ensures consistency and reduces code duplication

4.1. Unary Plus (+) and Minus (-)

These usually return a new object (not modify the original).

As member function:

```
class Complex {
    double re, im;
public:
    Complex(double r = 0, double i = 0) : re(r), im(i) {}
    // Unary minus (member)
    Complex operator-() const {
        return Complex(-re, -im);
    }
    // Unary plus (often unnecessary, but possible)
    Complex operator+() const {
        return *this; // Returns copy
    }
};
```

4.1. Unary Plus (+) and Minus (-)

As non-member function (rarely needed, but possible):

```
friend Complex operator-(const Complex& c) {
    return Complex(-c.re, -c.im);
}
```

✅ Member form is sufficient and common.

4.2. Increment and Decrement: ++, --

Two forms:

- Pre-increment: ++x → returns reference to modified object
- Post-increment: x++ → returns copy of original value

As member functions (standard approach):

```
class Counter {
    int value;
public:
    Counter(int v = 0) : value(v) {}

    // Pre-increment: ++x
    Counter& operator++() {
        ++value;
        return *this; // return reference to *this
    } // …
```

4.2. Increment and Decrement: ++, --

```
// …
    // Post-increment: x++
    Counter operator++(int) { // dummy 'int' parameter
                              // distinguishes post-
                              // increment
        Counter temp = *this; // save current state
        ++*this;
        return temp;      // return copy
    }
};
```

⚠ Never implement these as non-member functions unless you have a very good reason.

5. Binary Arithmetic Operators

5.1. Compound Assignment Operators (+=, -=, etc.)

These modify the left operand → ideal as member functions.

```
class Vector {
    double x, y;
public:
    Vector(double x = 0, double y = 0) : x(x), y(y) {}
    Vector& operator+=(const Vector& other) {
        x += other.x;
        y += other.y;
        return *this; // enable chaining: a += b += c;
    }
};
```

✅ Always return *this by reference.

5. Binary Arithmetic Operators

These create a new object → best implemented as non-member functions.

Recommended pattern: build on compound assignment

```
// Non-member binary + (uses +=)
Vector operator+(Vector lhs, const Vector& rhs) {
    lhs += rhs; // lhs is a copy → safe to
                // modify
    return lhs;
}
```

 Why pass lhs by value?

→ It creates a copy automatically, so we can modify it and return it efficiently (thanks to move semantics in C++11+).

5. Binary Arithmetic Operators

Alternative: pass both by `const&` and construct new object

```
Vector operator+(const Vector& a, const Vector& b) {
    return Vector(a.x + b.x, a.y + b.y);
}
```

✅ Also valid, but doesn't reuse += logic.

Or

```
Vector operator+(const Vector& a, const Vector& b) {
    auto tmp = a;
    return tmp += b;
}
```

5. Binary Arithmetic Operators

5.3. Mixed-Type Operations (e.g., Vector + double)

Non-member functions allow symmetry and mixed types:

```
// Vector * scalar
Vector operator*(Vector v, double scalar) {
    v.x *= scalar;
    v.y *= scalar;
    return v;
}
// scalar * Vector (enabled only by non-member!)
Vector operator*(double scalar, const Vector& v) {
    return Vector(v.x * scalar, v.y * scalar);
}
```

OPERATOR	RECOMMENDED FORM	RETURN TYPE	NOTES
`+` , `-` (unary)	Member	New object (by value)	`const`
`++` , `--` (pre)	Member	`Class&`	Modify `*this`
`++` , `--` (post)	Member	`Class` (copy)	Dummy `int` param
`+=` , `-=` , `*=` , `/=`	Member	`Class&`	Return `*this`
`+` , `-` , `*` , `/` (binary)	**Non-member**	New object (by value)	Reuse compound assignment
Mixed-type ops	**Non-member**	New object	Enables symmetry

6. Best Practices Recap

1. Compound assignments (+=) → member functions
2. Basic binary operators (+) → non-member functions
3. Implement + in terms of += (and - in terms of += and unary -)
4. Use const correctness for read-only operations
5. Enable mixed-type operations with non-member overloads
6. Avoid overloading just for fun — preserve intuitive meaning

Overloading Comparison Operators in C++

1. Introduction

Comparison operators (==, !=, <, >, <=, >=) are fundamental for ordering, searching, and decision-making in programs. When you define your own classes, you can overload these operators to allow natural comparisons:

```
Person a("Alice", 25), b("Bob", 30);
if (a == b) { /* ... */ }          // == operator
if (a < b)  { /* ... */ }          // < operator
if (a != b) { /* ... */ }          // != operator
```

In this lecture, we'll explore how to overload comparison operators correctly and efficiently.

2. Which Comparison Operators to Overload?

The “Big Three” to Start With:

- operator== — equality (most fundamental)
- operator< — less-than (for ordering)
- operator!= — inequality (often derived from ==)

✅ Once you have == and <, you can derive the rest (>, <=, >=).

3. Equality: `operator==`

✅ Recommended: Non-member function (often as friend)

```
class Point {
    double x, y;
public:
    Point(double x = 0, double y = 0) : x(x), y(y) {}

    // Friend declaration allows access to private members
    friend bool operator==(const Point& lhs, const Point& rhs);
};
bool operator==(const Point& lhs, const Point& rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
```

🔑 Always pass by const& to avoid copying and to allow comparing const objects.

3. Equality: `operator==`

✅ Derive `operator!=` from `operator==`:

```
bool operator!=(const Point& lhs, const Point&
rhs) {
    return !(lhs == rhs);  // Reuse ==
}
```

✅ This is more maintainable: only one logic to update.

4. Ordering: `operator<`

This is crucial for sorting and associative containers like std::set, std::map.

✅ Recommended: Non-member function

```
bool operator<(const Point& lhs, const Point& rhs) {
    if (lhs.x != rhs.x) {
        return lhs.x < rhs.x;  // Compare x first
    }
    return lhs.y < rhs.y;      // Then compare y
}
```

🔍 This is a lexicographic (dictionary-style) ordering.

4. Ordering: `operator<`

- ✅ **Derive other comparison operators:**

```
bool operator>(const Point& lhs, const Point& rhs) {
    return rhs < lhs;  // Reverse operands
}
bool operator<=(const Point& lhs, const Point& rhs) {
    return !(rhs < lhs);  // Not (rhs > lhs)
}
bool operator>=(const Point& lhs, const Point& rhs) {
    return !(lhs < rhs);  // Not (lhs < rhs)
}
```

Overloading Stream Insertion (<<) and Extraction (>>) Operators in C++

1. Introduction

In C++, input and output are handled through streams:

- Output: `std::cout, std::ofstream, std::ostringstream` → use insertion operator <<
- Input: `std::cin, std::ifstream, std::istringstream` → use extraction operator >>

By default, these operators work with built-in types (int, string, etc.).

But what if you want to print or read your own class?

```
Person p("Alice", 30);
std::cout << p;  // ← How can this work?
std::cin >> p;   // ← And this?
```

✅ The answer: overload `operator<<` and `operator>>`!

2. Why Overload Stream Operators?

- Enable natural I/O syntax for custom types
- Integrate your classes with the C++ I/O system
- Make debugging and logging easier and cleaner
- Support file serialization and parsing

⚠ Important: These operators must be overloaded as non-member functions, because the left operand is a stream (`std::ostream` or `std::istream`), not your class.

3. Overloading `operator<<` (Stream Insertion)

Goal:

Allow `std::cout << myObject;`

Syntax:

```
std::ostream& operator<<(std::ostream& os, const MyClass& obj);
```

Key Rules:

- Return std::ostream& → enables chaining: cout << a << b << endl;
- Pass object by const reference → avoids copying and doesn't modify it
- Declare as friend if accessing private members

3. Overloading `operator<<` (Stream Insertion)

✅ Example: Person Class

```
#include <iostream>
#include <string>

class Person {
    std::string name;
    int age;
public:
    Person(const std::string& n = "", int a = 0) : name(n), age(a) {}
    // Friend declaration allows access to private members
    friend std::ostream& operator<<(std::ostream& os, const Person& p);
};

// Definition of operator<<
std::ostream& operator<<(std::ostream& os, const Person& p) {
    os << "Name: " << p.name << ", Age: " << p.age;
    return os;  // Crucial for chaining!
}
```

Usage:

```
Person alice("Alice", 28);
std::cout << alice << std::endl;
// Output: Name: Alice, Age: 28
```

4. Overloading operator>> (Stream Extraction)

Goal:

Allow `std::cin >> myObject;`

Syntax:

```
std::istream& operator>>(std::istream& is, MyClass& obj);
```

Key Rules:

- Return std::istream& → enables chaining: cin >> a >> b;
- Pass object by non-const reference → you will modify it
- Do not print prompts inside operator>> (e.g., no "Enter name: ") — keep it pure
- Often declared as friend

4. Overloading operator>> (Stream Extraction)

✅ Example: Extending Person

```
class Person {
    // ... (same as before)

    // Add friend for input
    friend std::istream& operator>>(std::istream& is, Person& p);
};

std::istream& operator>>(std::istream& is, Person& p) {
    is >> p.name >> p.age;
    return is;
}
```

🔍 This assumes input format: Alice 28 (space-separated)

4. Overloading operator>> (Stream Extraction)

Usage:

```
Person p;
std::cout << "Enter name and age: ";
std::cin >> p;  // User types: Bob 35

std::cout << "You entered: " << p << std::endl;
// Output: Name: Bob, Age: 35
```

5. Handling Complex Input Formats

Sometimes data aren’t space-separated. Example: "Alice,28"

Custom Parsing in operator>>:

```
std::istream& operator>>(std::istream& is, Person& p) {
    std::string line;
    if (std::getline(is, line)) {
        size_t comma = line.find(',');
        if (comma != std::string::npos) {
            p.name = line.substr(0, comma);
            p.age = std::stoi(line.substr(comma + 1));
        }
    }
    return is;
}
```

Now input can be:

```
Alice,28
```

💡 Tip: For file I/O, this allows reading CSV-like formats easily.

6. Best Practices

RULE	REASON
Always return the stream	Enables `cout << a << b;`
Use const for <<	Don’t modify the object when printing
Don’t use `const` for >>	You need to modify the object
Prefer friend only when needed	If you can use public getters/setters, do so
Don’t include prompts in >>	Keep I/O operators pure and reusable
Handle input errors gracefully	Check stream state if needed

7. Full Working Example

```
#include <iostream>
class Point {
    double x, y;
public:
    Point(double x = 0, double y = 0) : x(x), y(y) {}
    // Output: (x, y)
    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
    // Input: x y (space-separated)
    friend std::istream& operator>>(std::istream& is, Point& p) {
        is >> p.x >> p.y;
        return is;
    }
};
```

7. Full Working Example

```
int main() {
    Point p1(1.5, -2.0);
    std::cout << "Point: " << p1 << std::endl;
    // (1.5, -2)
    Point p2;
    std::cout << "Enter x y: ";
    std::cin >> p2;
    std::cout << "You entered: " << p2 << std::endl;
    return 0;
}
```

8. Common Mistakes to Avoid

❌ Returning void

```
void operator<<(std::ostream& os, const MyClass& obj);
// ❌ breaks chaining!
```

❌ Making it a member function

```
class MyClass {
public:
    std::ostream& operator<<(std::ostream& os);
 // ❌ left operand is *this (MyClass), not ostream!
};
```

❌ Forgetting const in `operator<<`

```
std::ostream& operator<<(std::ostream& os, MyClass&
obj); // ❌ can't print const objects!
```

9. Summary

OPERATOR	PURPOSE	SIGNARURE	MUST BE NON-MEMBER
<<	Output (insertion)	`ostream& operator<<(ostream&, const T&)`	✅ Yes
>>	Input (extraction)	`istream& operator>>(istream&, T&)`	✅ Yes

By overloading these operators:

- Your classes integrate seamlessly with C++ I/O
- Code becomes cleaner and more readable
- You enable serialization, logging, and user interaction

Remember:

“Make your types behave like built-in types.”

— Stroustrup

Now you can cout << yourObject as naturally as cout << 42!

Singly Linked Lists in C++

Using a Template Node Structure and Solving Common Problems

1. Introduction

In this lecture, we'll explore singly linked lists in C++ without wrapping them in a class. Instead, we'll work directly with:

- A templated Node structure
- Free functions to manipulate the list
- Raw pointers and manual memory management

This approach helps you:

- Understand the core mechanics of linked lists
- Practice pointer manipulation
- Solve classic interview-style problems

We'll also use templates so our list can store any data type (int, string, double, etc.).

2. What Is a Singly Linked List?

A singly linked list is a linear collection of elements called nodes.

Each node contains:

- Data (e.g., an integer, string, or object)
- A pointer to the next node in the sequence

The last node

[Data | Next] → [Data | Next] → [Data | Next] → nullptr

↑

Head (points to first node)

points to `nullptr`, marking the end of the list.

🔑 Key idea: No random access — you must traverse from the head to reach any node.

2. The Node Structure (Templated)

We define a generic Node that can hold any type of data:

```
template <typename T>
struct Node {
    T data;
    Node<T>* next;
    // Constructor
    Node(const T& value) : data(value),
next(nullptr) {}
};
```

✅ Node<int>, Node<std::string>, etc., will be generated automatically by the compiler.

3. Basic Operations as Free Functions

Since we're not using a class, the head pointer is managed by the user and passed to functions by reference (so we can modify it).

3.1. Insert at Front

```
template <typename T>
void push_front(Node<T>*& head, const T& value) {
    Node<T>* newNode = new Node<T>(value);
    newNode->next = head;
    head = newNode;
}
```

🔑 Note: `Node<T>*& head` — we pass the pointer by reference to update the original head.

3. Basic Operations as Free Functions

3.2. Print the List

```
template <typename T>
void print_list(const Node<T>* head) {
    while (head) {
        std::cout << current->data << " -> ";
        head = head->next;
    }
    std::cout << "nullptr\n";
}
```

🔒 const ensures we don’t accidentally modify the list.

3. Basic Operations as Free Functions

3.3. Delete the Entire List

```
template <typename T>
void delete_list(Node<T>*& head) {
    while (head != nullptr) {
        Node<T>* temp = head;
        head = head->next;
        delete temp;
    }
    // head is now nullptr
}
```

✅ Always clean up to avoid memory leaks.

4. Example: Basic Usage

```
#include <iostream>
#include <string>

int main() {
    Node<int>* head = nullptr;  // Start with empty list
    push_front(head, 30);
    push_front(head, 20);
    push_front(head, 10);
    print_list(head);  // Output: 10 → 20 → 30 → nullptr
    delete_list(head);  // Clean up
    std::cout << "List deleted. head is "
              << (head ? "not null" : "nullptr") << std::endl;
    return 0;
}
```

5. Solving Typical Problems

Let’s implement solutions to common linked list problems using our template-based approach.

Problem 1: Find the Length of the List

Goal: Return the number of nodes.

```
template <typename T>
int get_length(const Node<T>* head) {
    int count = 0;
    while (head != nullptr) {
        count++;
        head = head->next;
    }
    return count;
}
Usage:
std::cout << "Length: " << get_length(head) << std::endl; // e.g., 3
```

Problem 2: Search for a Value

Goal: Return true if value exists.

```
template <typename T>
bool search(const Node<T>* head, const T& value) {
    while (head != nullptr) {
        if (head->data == value) {
            return true;
        }
        head = head->next;
    }
    return false;
}
```

Usage:

```
if (search(head, 20)) { std::cout << "Found 20!\n"; }
```

Problem 3: Insert at the End

Goal: Add a new node at the tail.

```
template <typename T>
void push_back(Node<T>*& head, const T& value) {
    Node<T>* newNode = new Node<T>(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node<T>* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = newNode;
}
```

 Time complexity: O(n)

Problem 4: Delete the First Occurrence of a Value

Goal: Remove one node with given value (if exists).

```
template <typename T>
void delete_value(Node<T>*& head, const T& value) {
    // Case 1: empty list
    if (head == nullptr) return;
    // Case 2: delete head
    if (head->data == value) {
        Node<T>* temp = head;
        head = head->next;
        delete temp;
        return;
    }
    // Case 3: delete in the middle or end
    Node<T>* current = head;
    while (current->next != nullptr && current->next->data != value) current = current->next;
    if (current->next != nullptr) { // Found it
        Node<T>* toDelete = current->next;
        current->next = toDelete->next;
        delete toDelete;
    }
}
```

Problem 5: Reverse the List (In-Place)

Goal: Reverse the direction of pointers.

```
template <typename T>
void reverse_list(Node<T>*& head) {
    Node<T>* prev = nullptr;
    Node<T>* current = head;
    Node<T>* next = nullptr;
    while (current != nullptr) {
        next = current->next;   // Save next
        current->next = prev;   // Reverse link
        prev = current;         // Move prev forward
        current = next;         // Move current forward
    }
    head = prev;  // New head is the last node
}
```

✅ This is a classic interview question!

7. Key Takeaways

CONCEPT	WHY IT MATTERS
Templated Node	Reusable for any data type
Pass head by reference	Allows functions to change which node is the head
Manual memory management	You must `delete` every `new`
Pointer traversal	Core skill for linked structures
Edge cases	Always test: empty list, single node, head/tail operations

8. Common Pitfalls

- ❌ Forgetting to initialize head to `nullptr`
- ❌ Passing head by value → changes don’t persist
- ❌ Not handling empty list in deletion/search
- ❌ Memory leaks from missing delete

✅ Always test with:

- Empty list
- One element
- Two elements
- Operation on head/tail

9. What’s Next?

Once comfortable with raw pointers and free functions, you can:

- Wrap everything in a LinkedList<T> class
- Add iterators
- Implement copy constructor and assignment operator
- Compare performance with std::forward_list

10. Practice Problems

- Write a function to find the middle node (use slow/fast pointers).
- Check if the list is a palindrome.
- Detect a cycle in the list (Floyd’s cycle-finding algorithm).
- Merge two sorted lists into one sorted list.

💡 These are common in technical interviews!