كليـــة العلـــــوم  
قسم الأمن السيبراني

**Subject: Object Oriented Programming (OOP)**

**Second Stage**

**Lecturer: Dr. Abdulkadhem A. Abdulkadhem**

**Lecture (11)**

**Operator Overloading**

This lecture will focus on the concept of operator overloading in C++. By the end of this lecture, you will understand how to redefine standard operators such as **+, -, and ==** to work with user-defined types like classes and objects.

**1. Introduction to Operator Overloading**

Operator overloading allows you to redefine the way operators work for user-defined types. For instance, you can define how the **+ operator** adds two objects of a class or how the == operator compares them.

* **Why is it useful?**
  + To enhance readability and expressiveness in code.
  + To enable object-oriented behavior for standard operators.

**2. Overloading Unary Operators**

Unary operators are those that operate on a single operand, such as **++, --, or -.**

**Steps to Overload Unary Operators**

1. Define the operator as a member function or a friend function.
2. Ensure the operator takes no arguments.
3. Use the operator keyword followed by the operator symbol.

**Example: Overloading the - Operator**

|  |
| --- |
| #include <iostream>  using namespace std;  class Number {  int value;  public:  Number(int v) : value(v) {}  // Overloading the unary minus (-) operator  **Number operator-()** {  return Number(-value);  }  void display() {  cout << "Value: " << value << endl;  }  };  int main() {  Number n(5);  Number neg = -n; // Calls overloaded operator-  neg.display(); // Output: Value: -5  return 0;  } |

**3. Operator Arguments, Return Values, and Postfix Notation**

**Operator Arguments**

Some operators, like binary operators, require arguments to function. For example, the + operator will need two operands.

**Return Values**

An overloaded operator can return:

* A modified object.
* A value of a primitive type (e.g., int, bool).

**Prefix vs. Postfix Notation**

For postfix operators like x++, the compiler uses an additional int parameter to distinguish it from prefix ++x. The *int* inside the parentheses does not represent an integer value being passed to the function. Instead, it is a **dummy argument** used to distinguish the postfix increment operator from the prefix increment operator which takes no arguments. The presence of this *int* tells the compiler that this is the postfix version of the ++ operator.

**Example: Overloading the Prefix vs. Postfix**

|  |
| --- |
| #include <iostream>  using namespace std;  class Counter {  private:  int count; // Counter value  public:  // Constructor to initialize counter  Counter(int c) : count(c) {}  // Prefix increment (++obj)  Counter operator++() {  return Counter(++count); // Increment first, then return  }  // Postfix increment (obj++)  Counter operator++(int) {  return Counter(count++); // Return current value, then increment  }  // Function to print the counter value  void print() const {  cout << count << endl;  }  };  int main() {  Counter c1(1), c2(1); // Initialize two Counter objects with value 1  // Demonstrate prefix increment  c2 = ++c1; // c1=2, c2=2 (increment c1 first, then assign to c2)  cout << "After prefix increment:" << endl;  cout << "c1: ";  c1.print(); // c1 = 2  cout << "c2: ";  c2.print(); // c2 = 2  // Demonstrate postfix increment  c2 = c1++; // c1=3, c2=2 (assign c1 to c2, then increment c1)  cout << "After postfix increment:" << endl;  cout << "c1: ";  c1.print(); // c1 = 3  cout << "c2: ";  c2.print(); // c2 = 2  return 0;  } |

The output:

*After prefix increment:*

*c1: 2*

*c2: 2*

*After postfix increment:*

*c1: 3*

*c2: 2*

**4. Overloading Binary Operators**

Binary operators operate on two operands. Examples include **+, -, \*, /,** **and comparison operators like == or <.**

**Steps to Overload Binary Operators**

1. Define the operator as a member function or a friend function.
2. Pass the second operand as an argument.
3. Return the result of the operation.

**Example: Overloading the + Operator**

|  |
| --- |
| #include <iostream>  using namespace std;  class Complex {  float real, imag;  public:  Complex(float r, float i) : real(r), imag(i) {}  // Overloading the + operator  **Complex operator+(const Complex& other)** {  return Complex(real + other.real, imag + other.imag);  }  void display() {  cout << real << " + " << imag << "i" << endl;  }  };  int main() {  Complex c1(2.5, 3.5), c2(1.5, 4.5);  Complex sum = c1 + c2; // Calls overloaded operator+  sum.display(); // Output: 4.0 + 8.0i  return 0;  } |

**5. Overloading Arithmetic Operators**

Arithmetic operators include +, -, \*, /, etc. These can be overloaded to perform operations on objects of a class.

**Example: Overloading the \* Operator for a Matrix Class**

|  |
| --- |
| #include <iostream>  using namespace std;  class Matrix {  int value;  public:  Matrix(int v) : value(v) {}  **Matrix operator\*(const Matrix& other)** {  return Matrix(value \* other.value);  }  void display() {  cout << "Result: " << value << endl;  }  };  int main() {  Matrix m1(3), m2(4);  Matrix result = m1 \* m2; // Calls overloaded operator\*  result.display(); // Output: Result: 12  return 0;  } |

**6. Overloading Comparison Operators**

Comparison operators like ==, <, and > can also be overloaded to compare objects based on their data members.

**Example: Overloading the == Operator**

|  |
| --- |
| #include <iostream>  using namespace std;  class Point {  int x, y;  public:  Point(int x, int y) : x(x), y(y) {}  **bool operator==(const Point& other)** {  return (x == other.x) && (y == other.y);  }  };  int main() {  Point p1(2, 3), p2(2, 3);  if (p1 == p2) { // Calls overloaded operator==  cout << "Points are equal" << endl;  } else {  cout << "Points are not equal" << endl;  }  return 0;  } |

**Unary Operators:** Unary operators operate on a single operand.

| **Operator** | **Purpose** | **Example** |
| --- | --- | --- |
| + | Unary plus (typically redundant, but can be used for marking positive values explicitly). | +a |
| - | Unary minus (used to negate a value). | -a |
| \* | Dereference operator (used to access the value at a memory address). | \*ptr |
| ! | Logical NOT (inverts a boolean value). | !a |
| ~ | Bitwise NOT (inverts all bits of a number). | ~a |
| & | Address-of operator (returns the memory address of a variable). | &a |
| ++ | Increment operator (adds 1 to a variable). | ++a (prefix) or a++ (postfix) |
| -- | Decrement operator (subtracts 1 from a variable). | --a (prefix) or a-- (postfix) |
| () | Function call operator (can be overloaded to make an object callable like a function). | object() |
| -> | Member access operator (used to access class members via a pointer). | ptr->member |
| ->\* | Pointer-to-member operator (used to access members of a class via a pointer to member). | ptr->\*memberPointer |
| new | Memory allocation operator (allocates memory dynamically). | int \*p = new int; |
| delete | Memory deallocation operator (frees dynamically allocated memory). | delete p; |

**Binary Operators:** Binary operators operate on two operands.

| **Operator** | **Purpose** | **Example** |
| --- | --- | --- |
| + | Addition operator. | a + b |
| - | Subtraction operator. | a - b |
| \* | Multiplication operator. | a \* b |
| / | Division operator. | a / b |
| % | Modulus operator (remainder of division). | a % b |
| & | Bitwise AND operator. | a & b |
| ` | ` | Bitwise OR operator. |
| ^ | Bitwise XOR operator. | a ^ b |
| << | Left shift operator (shifts bits to the left). | a << b |
| >> | Right shift operator (shifts bits to the right). | a >> b |
| += | Add and assign operator. | a += b (equivalent to a = a + b) |
| -= | Subtract and assign operator. | a -= b (equivalent to a = a - b) |
| /= | Divide and assign operator. | a /= b (equivalent to a = a / b) |
| %= | Modulus and assign operator. | a %= b (equivalent to a = a % b) |
| &= | Bitwise AND and assign operator. | a &= b (equivalent to a = a & b) |
| ^= | Bitwise XOR and assign operator. | a ^= b (equivalent to a = a ^ b) |
| <<= | Left shift and assign operator. | a <<= b (equivalent to a = a << b) |
| >>= | Right shift and assign operator. | a >>= b (equivalent to a = a >> b) |
| == | Equality comparison operator. | a == b (true if a equals b) |
| != | Inequality comparison operator. | a != b (true if a is not equal to b) |
| < | Less-than operator. | a < b |
| > | Greater-than operator. | a > b |
| <= | Less-than-or-equal-to operator. | a <= b |
| >= | Greater-than-or-equal-to operator. | a >= b |
| && | Logical AND operator. | a && b (true if both a and b are true) |
| [] | Subscript operator (used for accessing array elements or containers). | array[index] |
| () | Function call operator (allows an object to be treated as a function). | object(params) |

**Operators That Cannot Be Overloaded**

| **Operator** | **Description** |
| --- | --- |
| . | Dot operator |
| .\* (or ->) | Access member operator |
| :: | Scope resolution operator |
| ?: | Conditional operator (ternary) |
| sizeof | Size of file or type operator |

**7. Summary of Key Points**

* Operator overloading enhances the usability of custom classes.
* Unary operators operate on a single object; binary operators require two operands.
* Always maintain the operator's expected behavior to avoid confusion.
* Operator overloading is commonly used for mathematical classes, string manipulation, and comparisons.

**Assignment for Students**

**Q: Overload the == operator for a Student class to compare two students based on their IDs.**

|  |
| --- |
| Hint:  **Define the Student Class:** Create a class named Student with the following attributes:   * id: A unique identifier for the student. * Any additional attributes like name, age, etc. (optional, for context).    **Overload the == Operator:** Implement the operator== function for the Student class. The purpose of this operator is to compare the id attribute of two Student objects and return true if their IDs are equal and false otherwise.   **Write a Main Function:**   * Create multiple objects of the Student class. * Use the == operator to compare these objects. * Print the results to verify that the overloaded operator works as intended |