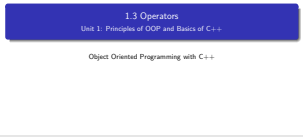
1.3 Operators

Unit 1: Principles of OOP and Basics of C++

Object Oriented Programming with C++



Lecture 7: C++ Operators

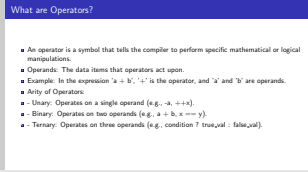
- Unit 1: Principles of OOP and Basics of C++
- Topic 1.3: Operators
- Instructor: [Your Name]
- Course: DI05011021 - Object Oriented Programming with C++

Lecture 7: C++ Operators
- Unit 1: Principles of OOP and Basics of C++
- Topic 1.3: Operators
- Instructor: [Your Name]
- Course: DI05011021 - Object Oriented Programming with C++

Welcome back everyone. Today we are diving into Lecture 7, which focuses entirely on C++ Operators. Now that we know how to store data in variables from our last lecture, we need to know how to manipulate that data. Operators are the tools we use to perform computations, comparisons, and logic in our code.

What are Operators?

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- Operands: The data items that operators act upon.
- Example: In the expression 'a + b', '+' is the operator, and 'a' and 'b' are operands.
- Arity of Operators:
- - Unary: Operates on a single operand (e.g., -a, ++x).
- - Binary: Operates on two operands (e.g., a + b, x == y).
- - Ternary: Operates on three operands (e.g., condition ? true_val : false_val).



Let's define what an operator actually is. It's simply a symbol that triggers a specific action, like addition or comparison. The variables or values it works on are called operands. A key concept here is 'arity', which just means how many operands an operator needs. Most operators you use every day, like plus or minus, are binary because they need two numbers. But some, like the negative sign in front of a single number, are unary.

1. Arithmetic Operators

- Used to perform common mathematical operations.
- Assume A = 10 and B = 20:
- - Addition (+): Adds two operands. (A + B = 30)
- - Subtraction (-): Subtracts second operand from the first. (A - B = -10)
- - Multiplication (*): *Multiplies both operands.* (A * B = 200)
- - Division (/): Divides numerator by numerator. (B / A = 2)
- - Modulus (%): Returns the remainder of an integer division. (B % A = 0)

1. Arithmetic Operators
- Used to perform common mathematical operations.
- Assume A = 10 and B = 20:
- - Addition (+): Adds two operands. (A + B = 30)
- - Subtraction (-): Subtracts second operand from the first. (A - B = -10)
- - Multiplication (*): *Multiplies both operands.* (A * B = 200)
- - Division (/): Divides numerator by numerator. (B / A = 2)
- - Modulus (%): Returns the remainder of an integer division. (B % A = 0)

Arithmetic operators are the most familiar. They map directly to basic math. Addition, subtraction, and multiplication are straightforward. Division in C++ has a quirk: if both operands are integers, it performs integer division, discarding any fractional part. The modulus operator, represented by the percent sign, is incredibly useful in programming; it gives you the remainder after division. For example, 5 modulus 2 is 1. It's often used to check if a number is even or odd.

2. Relational (Comparison) Operators

- Used to compare two values. They return a boolean value (true/1 or false/0).
- Assume A = 10 and B = 20:
- - Equal to (==): Checks if values are equal. (A == B is false)
- - Not equal to (!=): Checks if values are not equal. (A != B is true)
- - Greater than (¿): (A ¿ B is false)
- - Less than (¡): (A ¡ B is true)
- - Greater than or equal to (¿=): (A ¿= B is false)
- - Less than or equal to (¡=): (A ¡= B is true)

2. Relational (Comparison) Operators
- Used to compare two values. They return a boolean value (true/1 or false/0).
- Assume A = 10 and B = 20:
- - Equal to (==): Checks if values are equal. (A == B is false)
- - Not equal to (!=): Checks if values are not equal. (A != B is true)
- - Greater than (¿): (A ¿ B is false)
- - Less than (¡): (A ¡ B is true)
- - Greater than or equal to (¿=): (A ¿= B is false)
- - Less than or equal to (¡=): (A ¡= B is true)

When we need to make decisions in code, we compare things. Relational operators do exactly this. They always evaluate to a boolean result: either true or false. A very common beginner mistake is using a single equals sign '=' for comparison. Remember, a single equals is for assignment; double equals '==' is for checking equality. 'Not equal' uses the exclamation mark, which often means 'not' in programming.

3. Logical Operators

- Used to combine multiple conditions or reverse logical states.
- Assume A = 1 (true) and B = 0 (false):
- - Logical AND (&&): True only if BOTH operands are true. (A && B is false)
- - Logical OR (——): True if AT LEAST ONE operand is true. (A —— B is true)
- - Logical NOT (!): Reverses the logical state of its operand. (!A is false)

3. Logical Operators
- Used to combine multiple conditions or reverse logical states.
- Assume A = 1 (true) and B = 0 (false):
- - Logical AND (&&): True only if BOTH operands are true. (A && B is false)
- - Logical OR (——): True if AT LEAST ONE operand is true. (A —— B is true)
- - Logical NOT (!): Reverses the logical state of its operand. (!A is false)

Logical operators allow us to build complex conditions. The AND operator, double ampersand, requires everything to be true. The OR operator, double pipe, only needs one thing to be true. The NOT operator, the exclamation mark, flips true to false and false to true. These are fundamental when we start writing 'if' statements later in the course.

Deep Dive: Short-Circuit Evaluation

- C++ optimizes logical operations using short-circuiting.
- For Logical AND (&&):
- - If the first operand evaluates to false, the entire expression must be false.
- - C++ does not evaluate the second operand.
- For Logical OR (——):
- - If the first operand evaluates to true, the entire expression must be true.
- - C++ does not evaluate the second operand.
- Example: 'if (x != 0 && (10 / x) ¿ 1)' prevents divide-by-zero errors.

Deep Dive: Short-Circuit Evaluation
- C++ optimizes logical operations using short-circuiting.
- For Logical AND (&&):
- - If the first operand evaluates to false, the entire expression must be false.
- - C++ does not evaluate the second operand.
- For Logical OR (——):
- - If the first operand evaluates to true, the entire expression must be true.
- - C++ does not evaluate the second operand.
- Example: 'if (x != 0 && (10 / x) ¿ 1)' prevents divide-by-zero errors.

This is a very important technical detail called short-circuit evaluation. C++ is smart. If it's evaluating an AND expression and the first part is false, it already knows the whole thing will be false, so it skips executing the second part entirely. This isn't just for speed; it's a feature we can use. Look at the example on the slide. We check if x is not zero BEFORE we divide by x. Because of short-circuiting, if x IS zero, the division never happens, saving our program from crashing.

4. Bitwise Operators

- Perform operations on data at the bit (binary) level.
- Assume A = 60 (0011 1100) and B = 13 (0000 1101):
- - Bitwise AND (&): (A & B = 0000 1100, which is 12)
- - Bitwise OR (—): (A — B = 0011 1101, which is 61)
- - Bitwise XOR (ˆ): (A ˆ B = 0011 0001, which is 49)
- - Bitwise NOT (˜): Flips all bits. (˜A = 1100 0011)
- - Left Shift (¡¡): Shifts bits left, filling with 0. (A ¡¡ 2 = 1111 0000, which is 240)
- - Right Shift (¿¿): Shifts bits right. (A ¿¿ 2 = 0000 1111, which is 15)

4. Bitwise Operators
- Perform operations on data at the bit (binary) level.
- Assume A = 60 (0011 1100) and B = 13 (0000 1101):
- - Bitwise AND (&): (A & B = 0000 1100, which is 12)
- - Bitwise OR (—): (A — B = 0011 1101, which is 61)
- - Bitwise XOR (ˆ): (A ˆ B = 0011 0001, which is 49)
- - Bitwise NOT (˜): Flips all bits. (˜A = 1100 0011)
- - Left Shift (¡¡): Shifts bits left, filling with 0. (A ¡¡ 2 = 1111 0000, which is 240)
- - Right Shift (¿¿): Shifts bits right. (A ¿¿ 2 = 0000 1111, which is 15)

While arithmetic operators work on numbers as a whole, bitwise operators work on the individual 1s and 0s that make up those numbers in memory. These are often used in lower-level programming, embedded systems, or cryptography where memory and performance are strictly managed. Left shifting a number by 1 is essentially multiplying it by 2, and right shifting is dividing by 2, but done much faster at the hardware level.

5. Assignment Operators

- Used to assign values to variables.
- - Simple Assignment (=): Assigns right side operand to left side. (C = A + B)
- Compound (Shorthand) Assignments:
- - Add and Assign (+=): C += A is equivalent to C = C + A
- - Subtract and Assign (-=): C -= A is equivalent to C = C - A
- - Multiply and Assign (=): C = A is equivalent to C = C * A
- - Divide and Assign (/=): C /= A is equivalent to C = C / A
- - Modulus and Assign (%=): C %= A is equivalent to C = C % A

5. Assignment Operators
- Used to assign values to variables.
- - Simple Assignment (=): Assigns right side operand to left side. (C = A + B)
- Compound (Shorthand) Assignments:
- - Add and Assign (+=): C += A is equivalent to C = C + A
- - Subtract and Assign (-=): C -= A is equivalent to C = C - A
- - Multiply and Assign (=): C = A is equivalent to C = C * A
- - Divide and Assign (/=): C /= A is equivalent to C = C / A
- - Modulus and Assign (%=): C %= A is equivalent to C = C % A

We've used the basic assignment operator '=' already. It evaluates whatever is on the right side and stores it in the variable on the left side. To make code cleaner, C++ offers compound assignment operators. Instead of writing 'x = x + 5', you can simply write 'x += 5'. It means exactly the same thing to the compiler, but it's faster to type and easier to read.

6. Increment and Decrement Operators

- Unary operators to add or subtract 1 from a variable.
- - Increment (++): Increases integer value by 1. (A++ or ++A)
- - Decrement (–): Decreases integer value by 1. (A– or –A)
- Two forms: Prefix and Postfix.
- - Prefix (++A): Increment the value first, then use it in the expression.
- - Postfix (A++): Use the current value in the expression first, then increment it.

6. Increment and Decrement Operators
- Unary operators to add or subtract 1 from a variable.
- - Increment (++): Increases integer value by 1. (A++ or ++A)
- - Decrement (–): Decreases integer value by 1. (A– or –A)
- Two forms: Prefix and Postfix.
- - Prefix (++A): Increment the value first, then use it in the expression.
- - Postfix (A++): Use the current value in the expression first, then increment it.

Adding or subtracting one from a variable is so common in loops that C++ has special unary operators for it. '++' adds one, '–' subtracts one. But there's a catch: you can put the operator before the variable (prefix) or after it (postfix). While both add 1 to the variable eventually, they behave differently when used inside a larger equation.

Prefix vs. Postfix: A Closer Look

- Prefix Example:
- int x = 5;
- int y = ++x; // x becomes 6, then y is assigned 6.
- Result: x = 6, y = 6
-
- Postfix Example:
- int a = 5;
- int b = a++; // b is assigned 5, then a becomes 6.
- Result: a = 6, b = 5

Prefix vs. Postfix: A Closer Look
Prefix Example:
int x = 5;
int y = ++x; // x becomes 6, then y is assigned 6.
Result: x = 6, y = 6
Postfix Example:
int a = 5;
int b = a++; // b is assigned 5, then a becomes 6.
Result: a = 6, b = 5

Let's walk through this carefully. In the prefix example, '++x', we tell the computer: 'increment x first, then give me the new value to assign to y'. So x becomes 6, and y gets 6. In the postfix example, 'a++', we say: 'give me the current value of a to assign to b, and after that's done, increment a in the background'. So b gets the old value of 5, and then a becomes 6. This distinction is a very common source of subtle bugs for beginners.

Operator Precedence

- Determines the grouping of terms in an expression, affecting how it's evaluated.
- Similar to BODMAS/PEMDAS in mathematics.
- Example: 'int x = 7 + 3 * 2;'
- - Does x equal 20 ((7+3)*2) or 13 (7+(3*2))?
- - Multiplication has higher precedence than addition, so it's 13.
- Use parentheses () to override default precedence.
- Example: 'int x = (7 + 3) * 2;' makes x equal 20.

Operator Precedence
- Determines the grouping of terms in an expression, affecting how it's evaluated.
- Similar to BODMAS/PEMDAS in mathematics.
- Example: 'int x = 7 + 3 * 2;'
- - Does x equal 20 ((7+3)*2) or 13 (7+(3*2))?
- - Multiplication has higher precedence than addition, so it's 13.
- Use parentheses () to override default precedence.
- Example: 'int x = (7 + 3) * 2;' makes x equal 20.

When you have an expression with multiple operators, which one happens first? Just like in math class, C++ has an order of operations, called operator precedence. Multiplication happens before addition. If you want to force addition to happen first, you must use parentheses. Parentheses have the highest precedence of all. When in doubt, use parentheses to make your code's intent perfectly clear, even if you don't strictly need them.

Precedence Table (Partial)

- High Precedence:
- 1. () [] -¿ . (Function call, array subscript, member access)
- 2. ++ – ! ˜ - + (Unary operators)
- 3. * / % (Multiplicative)
- 4. + - (Additive)
- ...
- Low Precedence:
- n-2. && (Logical AND)
- n-1. —— (Logical OR)
- n. = += -= etc. (Assignment)

Precedence Table (Partial)
High Precedence:
1. () [] -¿ . (Function call, array subscript, member access)
2. ++ – ! ˜ - + (Unary operators)
3. * / % (Multiplicative)
4. + - (Additive)
...
Low Precedence:
n-2. && (Logical AND)
n-1. —— (Logical OR)
n. = += -= etc. (Assignment)

Here is a simplified look at the precedence table. You don't need to memorize the entire thing, but you should know the general tiers. Unary operators usually happen first, then multiplication/division, then addition/subtraction, then comparisons, then logical operators, and finally assignment. Notice that assignment has almost the lowest precedence. That makes sense, because you want the entire right side of the equation to finish calculating before you assign it to the variable on the left.

Operator Associativity

- What happens when operators have the SAME precedence?
- Associativity determines the direction of evaluation: Left-to-Right or Right-to-Left.
- Example: Left-to-Right (Most arithmetic/logical)
- - 'a - b + c' evaluates as '(a - b) + c'
- Example: Right-to-Left (Assignment, Unary)
- - 'a = b = c = 5' evaluates as 'a = (b = (c = 5))'
- - First c becomes 5, then b becomes 5, then a becomes 5.

Operator Associativity
- What happens when operators have the SAME precedence?
- Associativity determines the direction of evaluation: Left-to-Right or Right-to-Left.
- Example: Left-to-Right (Most arithmetic/logical)
- - 'a - b + c' evaluates as '(a - b) + c'
- Example: Right-to-Left (Assignment, Unary)
- - 'a = b = c = 5' evaluates as 'a = (b = (c = 5))'
- - First c becomes 5, then b becomes 5, then a becomes 5.

Precedence tells us what to do when operators are different. But what if they are the same, or on the same tier? Like addition and subtraction? That's where Associativity comes in. Most operators read left-to-right, just like reading a book. So 'a minus b plus c' is evaluated from the left. However, assignment operators are evaluated right-to-left. This allows chaining assignments like a = b = c = 5. It figures out the right-most part first, and cascades the values to the left.

Complex Expression Example

- Let's evaluate: int result = 5 + 2 * 3 - ++x;
- Assume x is initially 2.
- Step 1: Unary Prefix (++x) -¿ x becomes 3. Expression: 5 + 2 * 3 - 3
- Step 2: Multiplication (2 * 3) -¿ 6. Expression: 5 + 6 - 3
- Step 3: Addition (5 + 6) -¿ 11. Expression: 11 - 3 (Left-to-Right associativity)
- Step 4: Subtraction (11 - 3) -¿ 8.
- Step 5: Assignment (=) -¿ result becomes 8.

Complex Expression Example
- Let's evaluate: int result = 5 + 2 * 3 - ++x;
- Assume x is initially 2.
- Step 1: Unary Prefix (++x) -¿ x becomes 3. Expression: 5 + 2 * 3 - 3
- Step 2: Multiplication (2 * 3) -¿ 6. Expression: 5 + 6 - 3
- Step 3: Addition (5 + 6) -¿ 11. Expression: 11 - 3 (Left-to-Right associativity)
- Step 4: Subtraction (11 - 3) -¿ 8.
- Step 5: Assignment (=) -¿ result becomes 8.

Let's put precedence and associativity together in a complex example. Look at the expression on the slide. First, we find the operator with the highest precedence, which is the prefix increment '++x'. x becomes 3. Next highest is multiplication, so 2 times 3 is 6. Now we have addition and subtraction, which have the same precedence. So we look at associativity, which is left-to-right. We do 5 plus 6 to get 11. Finally, 11 minus 3 is 8. The very last step is the assignment operator, storing 8 into 'result'.

Summary & Q&A

- Operators allow us to perform data manipulation.
- Categorized into Arithmetic, Relational, Logical, Bitwise, Assignment, and Increment/Decrement.
- Understanding Prefix vs. Postfix is crucial for avoiding bugs.
- Precedence dictates order of operations; Associativity dictates direction when precedence is tied.
- Use parentheses () to make code clear and override default precedence.
- Any questions?

Summary & Q&A
- Operators allow us to perform data manipulation.
- Categorized into Arithmetic, Relational, Logical, Bitwise, Assignment, and Increment/Decrement.
- Understanding Prefix vs. Postfix is crucial for avoiding bugs.
- Precedence dictates order of operations; Associativity dictates direction when precedence is tied.
- Use parentheses () to make code clear and override default precedence.
- Any questions?

To summarize, operators are the verbs of our programming language, acting on the nouns (our variables). We've covered a lot of ground: math, comparisons, logic, and binary manipulation. Remember the golden rule of complex math in code: when in doubt, use parentheses. It prevents bugs and helps other programmers read your code. Are there any questions on anything we've covered today?