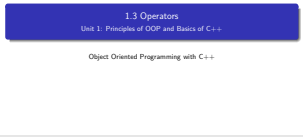
1.3 Operators

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

Object Oriented Programming with C++



Lecture 3: 1.3 Operators in C++

- Unit 1: Principles of OOP and Basics of C++
- Course: Object Oriented Programming with C++

Lecture 3: 1.3 Operators in C++
- Unit 1: Principles of OOP and Basics of C++
- Course: Object Oriented Programming with C++

Welcome everyone to Lecture 3. Today we are diving into a crucial part of any programming language: Operators. Operators are the building blocks that allow us to perform computations, make decisions, and manipulate data. We will cover a wide range of operators available in C++ and understand how the compiler decides which operation to perform first when multiple operators are combined.

What are Operators?

- Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.
- An operand is the data (variable or value) that the operator works on.
- Types based on the number of operands:
- 1. Unary Operators: Operate on a single operand (e.g., -a, a++).
- 2. Binary Operators: Operate on two operands (e.g., a + b, x == y).
- 3. Ternary Operator: Operates on three operands (e.g., condition ? true_val : false_val).

What are Operators?
- Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.
- An operand is the data (variable or value) that the operator works on.
- Types based on the number of operands:
- 1. Unary Operators: Operate on a single operand (e.g., -a, a++).
- 2. Binary Operators: Operate on two operands (e.g., a + b, x == y).
- 3. Ternary Operator: Operates on three operands (e.g., condition ? true_val : false_val).

Think of operators as verbs in the C++ language. They describe the action to be taken. The variables or values they act upon are the nouns, known as operands. In the expression '5 + 3', the '+' is the operator, while 5 and 3 are the operands. C++ has a rich set of operators, categorized mostly by how many operands they take. Binary operators are the most common, but unary operators are also frequently used for things like incrementing a counter.

Arithmetic Operators

- Used to perform mathematical operations on numerical data types.
- + (Addition): Adds two operands.
- - (Subtraction): Subtracts second operand from the first.
- * (Multiplication): Multiplies two operands.
- / (Division): Divides numerator by denominator.
- % (Modulo): Returns the remainder of an integer division.

Arithmetic Operators
- Used to perform mathematical operations on numerical data types.
- + (Addition): Adds two operands.
- - (Subtraction): Subtracts second operand from the first.
- * (Multiplication): Multiplies two operands.
- / (Division): Divides numerator by denominator.
- % (Modulo): Returns the remainder of an integer division.

Arithmetic operators are the most familiar since they map directly to basic math. However, there are a few C++ specific quirks to be aware of. The division operator '/' behaves differently depending on whether the operands are integers or floating-point numbers. If both are integers, it performs integer division, discarding any fractional part. The modulo operator '%' is incredibly useful in programming; it gives you the remainder of a division and only works with integer types. It's often used to check if a number is even or odd, or to wrap around values within an array bounds.

Arithmetic Operators in Action

- int a = 10, b = 3;
- int sum = a + b; // sum is 13
- int diff = a - b; // diff is 7
- int prod = a * b; // prod is 30
- int quot = a / b; // quot is 3 (Integer division!)
- int rem = a % b; // rem is 1 (Remainder of 10/3)
-
- float f_quot = (float)a / b; // f_quot is 3.333... (Float division)

Arithmetic Operators in Action
- int a = 10, b = 3;
- int sum = a + b; // sum is 13
- int diff = a - b; // diff is 7
- int prod = a * b; // prod is 30
- int quot = a / b; // quot is 3 (Integer division!)
- int rem = a % b; // rem is 1 (Remainder of 10/3)
-
- float f_quot = (float)a / b; // f_quot is 3.333... (Float division)

Let's look at some code. Notice the division operation: 10 divided by 3 is 3.333..., but because both 'a' and 'b' are initialized as integers, C++ truncates the decimal part, resulting in exactly 3. If we want the fractional result, we must cast at least one operand to a float, as shown in the last line. The modulo operator gives us 1, because 3 goes into 10 three times with a remainder of 1.

Increment and Decrement Operators

- Unary operators used to increase or decrease a variable's value by 1.
- ++ (Increment): Increases value by 1.
- – (Decrement): Decreases value by 1.
- Can be used as a Prefix (++a) or Postfix (a++).
- Prefix: Changes the value BEFORE evaluating the expression.
- Postfix: Evaluates the expression, THEN changes the value in the background.

Increment and Decrement Operators
- Unary operators used to increase or decrease a variable's value by 1.
- ++ (Increment): Increases value by 1.
- – (Decrement): Decreases value by 1.
- Can be used as a Prefix (++a) or Postfix (a++).
- Prefix: Changes the value BEFORE evaluating the expression.
- Postfix: Evaluates the expression, THEN changes the value in the background.

Increment and decrement operators are convenient shorthands for adding or subtracting 1. They are heavily used in loops, such as 'for' and 'while' loops. The distinction between prefix and postfix is a classic source of bugs for beginners. Prefix means 'update the value, then use it in the current statement'. Postfix means 'use the current value in this statement, then update it immediately after'.

Pre-increment vs Post-increment

- // Prefix Example
- int x = 5;
- int y = ++x; // Prefix increment
- // Result: x is now 6, y is 6
-
- // Postfix Example
- int a = 5;
- int b = a++; // Postfix increment
- // Result: a is now 6, b is 5

Pre-increment vs Post-increment
- // Prefix Example
- int x = 5;
- int y = ++x; // Prefix increment
- // Result: x is now 6, y is 6
-
- // Postfix Example
- int a = 5;
- int b = a++; // Postfix increment
- // Result: a is now 6, b is 5

This slide clearly illustrates the timing difference. In the first block, ++x is evaluated first. 'x' becomes 6, and then that new value (6) is assigned to 'y'. Both end up as 6. In the second block, the current value of 'a' (which is 5) is assigned to 'b' first. Only after the assignment is complete does 'a' increment to 6. Therefore, 'b' holds the old value. Understanding this difference is vital for writing correct loop logic.

Relational Operators

- Used to compare two values.
- They always return a boolean value: true (1) or false (0).
- == (Equal to): Checks if both operands are strictly equal.
- != (Not equal to): Checks if operands are different.
- ¿ (Greater than)
- ¡ (Less than)
- ¿= (Greater than or equal to)
- ¡= (Less than or equal to)

Relational Operators
- Used to compare two values.
- They always return a boolean value: true (1) or false (0).
- == (Equal to): Checks if both operands are strictly equal.
- != (Not equal to): Checks if operands are different.
- ¿ (Greater than)
- ¡ (Less than)
- ¿= (Greater than or equal to)
- ¡= (Less than or equal to)

Relational operators form the foundation of decision-making in C++. Whenever you need your program to choose between different paths based on a condition, you will use these. In C++, 'true' is typically represented as 1 and 'false' as 0 in memory. A very common beginner mistake is using a single equals sign '=' for comparison instead of the double equals '=='. Remember, a single '=' assigns a value, while double '==' compares values!

Relational Operators Example

- int p = 10, q = 20;
- bool r1 = (p == q); // r1 is false (0)
- bool r2 = (p != q); // r2 is true (1)
- bool r3 = (p ¡ q); // r3 is true (1)
- bool r4 = (p ¿= 10); // r4 is true (1)

Relational Operators Example
- int p = 10, q = 20;
- bool r1 = (p == q); // r1 is false (0)
- bool r2 = (p != q); // r2 is true (1)
- bool r3 = (p ¡ q); // r3 is true (1)
- bool r4 = (p ¿= 10); // r4 is true (1)

Here we see relational operators assessing conditions and assigning the resulting boolean values to variables. Notice how we use parentheses around the condition. While not strictly required here due to operator precedence, they make the code much more readable. These boolean results are what you will eventually place directly inside 'if' statements and loop controls.

Logical Operators

- Used to combine multiple relational expressions or boolean values.
- && (Logical AND): True ONLY if BOTH operands/conditions are true.
- —— (Logical OR): True if AT LEAST ONE operand/condition is true.
- ! (Logical NOT): Unary operator; reverses the boolean state of its operand (true becomes false, false becomes true).

Logical Operators
- Used to combine multiple relational expressions or boolean values.
- && (Logical AND): True ONLY if BOTH operands/conditions are true.
- —— (Logical OR): True if AT LEAST ONE operand/condition is true.
- ! (Logical NOT): Unary operator; reverses the boolean state of its operand (true becomes false, false becomes true).

Logical operators allow us to build complex, multi-part conditions. For example, if you want to check if a number is between 1 and 10, you cannot just write '1 ¡ x ¡ 10' in C++. You have to split it into two distinct relational expressions and combine them with a logical AND: 'x ¿ 1 && x ¡ 10'. The NOT operator is useful for flipping a condition, such as checking if a pointer is NOT null, or a string is NOT empty.

Short-Circuit Evaluation

- C++ optimizes logical operations using a technique called short-circuiting.
- For 'A && B': If A is false, B is NEVER evaluated. (The whole expression must be false).
- For 'A —— B': If A is true, B is NEVER evaluated. (The whole expression must be true).
- Benefits of Short-Circuiting:
- - Improves performance by skipping unnecessary evaluations.
- - Prevents runtime errors (e.g., checking if a pointer is null before trying to read its data).

Short-Circuit Evaluation
- C++ optimizes logical operations using a technique called short-circuiting.
- For 'A && B': If A is false, B is NEVER evaluated. (The whole expression must be false).
- For 'A —— B': If A is true, B is NEVER evaluated. (The whole expression must be true).
- Benefits of Short-Circuiting:
- - Improves performance by skipping unnecessary evaluations.
- - Prevents runtime errors (e.g., checking if a pointer is null before trying to read its data).

Short-circuit evaluation is a critical concept for writing safe and efficient code. Imagine an expression like '(divisor != 0) && (10 / divisor ¿ 1)'. Short-circuiting saves you from a fatal divide-by-zero error. If divisor IS 0, the first part evaluates to false. The compiler knows that 'false && anything' will always be false, so it entirely skips the '10 / divisor' part, preventing your program from crashing.

Bitwise Operators

- Operate directly on the individual bits (0s and 1s) of integer operands.
- & (Bitwise AND): 1 if both corresponding bits are 1.
- — (Bitwise OR): 1 if at least one corresponding bit is 1.
- ^ (Bitwise XOR): 1 if corresponding bits are DIFFERENT.
- ~ (Bitwise NOT): Inverts all bits (1s to 0s, 0s to 1s).
- ¡¡ (Left Shift): Shifts bits left, filling with zeros (effectively multiplies by 2^n).
- ¿¿ (Right Shift): Shifts bits right (effectively divides by 2^n).

Bitwise Operators
- Operate directly on the individual bits (0s and 1s) of integer operands.
- & (Bitwise AND): 1 if both corresponding bits are 1.
- — (Bitwise OR): 1 if at least one corresponding bit is 1.
- ^ (Bitwise XOR): 1 if corresponding bits are DIFFERENT.
- ~ (Bitwise NOT): Inverts all bits (1s to 0s, 0s to 1s).
- ¡¡ (Left Shift): Shifts bits left, filling with zeros (effectively multiplies by 2^n).
- ¿¿ (Right Shift): Shifts bits right (effectively divides by 2^n).

Bitwise operators work at the lowest level of data representation: binary digits. While you might not use them every day in high-level web applications, they are essential for systems programming, game engine development, cryptography, and embedded systems where memory and performance are highly constrained. Left shifting a number by 1 is functionally equivalent to multiplying it by 2, and right shifting by 1 is like dividing by 2, but bitwise operations execute much faster on the CPU hardware.

Bitwise Operators Deep Dive

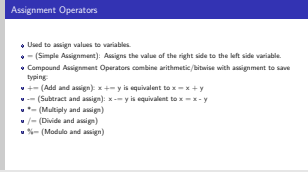
- int a = 5; // Binary: 0101
- int b = 3; // Binary: 0011
-
- int c = a & b; // 0001 (Decimal 1) - AND
- int d = a — b; // 0111 (Decimal 7) - OR
- int e = a ^ b; // 0110 (Decimal 6) - XOR
- int f = a ¡¡ 1; // 1010 (Decimal 10) - Shift Left by 1

Bitwise Operators Deep Dive
int a = 5; // Binary: 0101
int b = 3; // Binary: 0011
int c = a & b; // 0001 (Decimal 1) - AND
int d = a — b; // 0111 (Decimal 7) - OR
int e = a ^ b; // 0110 (Decimal 6) - XOR
int f = a ¡¡ 1; // 1010 (Decimal 10) - Shift Left by 1

This slide visually breaks down what happens during a bitwise operation. Take the bitwise AND (&). We line up the binary representations of 5 and 3. The AND operator looks down each vertical column. Only in the far rightmost column do both operands have a 1, so the result has a 1 there. The result is binary 0001, which is decimal 1. Left shifting 5 (0101) by 1 position pushes everything left, resulting in 1010, which evaluates to 10 in decimal.

Assignment Operators

- Used to assign values to variables.
- = (Simple Assignment): Assigns the value of the right side to the left side variable.
- Compound Assignment Operators combine arithmetic/bitwise with assignment to save typing:
- += (Add and assign): x += y is equivalent to x = x + y
- -= (Subtract and assign): x -= y is equivalent to x = x - y
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulo and assign)



The basic assignment operator is just the equals sign. Remember, data flows from right to left across the equals sign. C++ also provides compound assignment operators, which act as syntactical sugar. Writing 'x += y' does exactly the same thing as 'x = x + y', but it is shorter to type, less prone to typos, and arguably easier to read. These are universally used when updating counters or accumulating totals in loops.

Operator Precedence

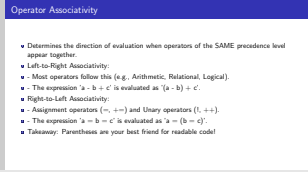
- Determines the exact order in which operators are evaluated within an expression.
- Similar to BODMAS/PEMDAS in standard mathematics.
- Operators with higher precedence are evaluated first.
- General Hierarchy (Highest to Lowest):
- 1. Parentheses () - Always overrides standard precedence
- 2. Postfix increment/decrement (a++)
- 3. Prefix increment/decrement, Logical NOT (++a, !a)
- 4. Multiplication, Division, Modulo (*, /, %)
- 5. Addition, Subtraction (+, -)
- 6. Relational (¡, ¿, ¡=, ¿=, ==, !=)
- 7. Logical AND (&&) then Logical OR (——)
- 8. Assignment (=, +=, -=)

Operator Precedence
- Determines the exact order in which operators are evaluated within an expression.
- Similar to BODMAS/PEMDAS in standard mathematics.
- Operators with higher precedence are evaluated first.
- General Hierarchy (Highest to Lowest):
- 1. Parentheses () - Always overrides standard precedence
- 2. Postfix increment/decrement (a++)
- 3. Prefix increment/decrement, Logical NOT (++a, !a)
- 4. Multiplication, Division, Modulo (*, /, %)
- 5. Addition, Subtraction (+, -)
- 6. Relational (¡, ¿, ¡=, ¿=, ==, !=)
- 7. Logical AND (&&) then Logical OR (——)
- 8. Assignment (=, +=, -=)

When an expression contains multiple different operators, how does the compiler know what to execute first? It uses strict operator precedence rules. Multiplication happens before addition, just like in math class. However, C++ has dozens of operators. If you want to force a specific order of evaluation, or if you are ever unsure, ALWAYS use parentheses. They have the highest precedence, they are practically free in terms of performance, and they make your code's intent unambiguously clear to human readers.

Operator Associativity

- Determines the direction of evaluation when operators of the SAME precedence level appear together.
- Left-to-Right Associativity:
- - Most operators follow this (e.g., Arithmetic, Relational, Logical).
- - The expression 'a - b + c' is evaluated as '(a - b) + c'.
- Right-to-Left Associativity:
- - Assignment operators (=, +=) and Unary operators (!, ++).
- - The expression 'a = b = c' is evaluated as 'a = (b = c)'.
- Takeaway: Parentheses are your best friend for readable code!



Precedence resolves ties between different operators, but what if you have two operators with the exact SAME precedence level? That's where associativity comes in. Subtraction and addition have the same precedence. Since they are left-to-right associative, 'a - b + c' is evaluated from left to right. Assignment, however, is right-to-left. In 'a = b = c', the value of 'c' is first assigned to 'b', and then the result of that assignment (which is 'b's new value) is subsequently assigned to 'a'.

Lecture Summary

- Operators act as the 'verbs' of C++, performing actions on operands.
- Arithmetic (+, -, *, /, %) perform math calculations.
- Increment/Decrement (++, –) adjust values; timing depends on prefix/postfix placement.
- Relational (==, !=, ¿, ¡) compare values, returning booleans.
- Logical (&&, ——, !) evaluate boolean logic, utilizing short-circuiting.
- Bitwise (&, —, ^, ¡¡) manipulate raw binary data.
- Assignment (=, +=) store computed values back into memory.
- Precedence and Associativity are the strict rules dictating execution order.

Lecture Summary
- Operators act as the 'verbs' of C++, performing actions on operands.
- Arithmetic (+, -, *, /, %) perform math calculations.
- Increment/Decrement (++, –) adjust values; timing depends on prefix/postfix placement.
- Relational (==, !=, ¿, ¡) compare values, returning booleans.
- Logical (&&, ——, !) evaluate boolean logic, utilizing short-circuiting.
- Bitwise (&, —, ^, ¡¡) manipulate raw binary data.
- Assignment (=, +=) store computed values back into memory.
- Precedence and Associativity are the strict rules dictating execution order.

To wrap up, operators are the fundamental tools in your C++ toolkit. We've covered a wide array today, from basic math to binary manipulation. The key to mastering them is practical application. Don't worry about memorizing the entire precedence table; you'll naturally learn the common ones over time. Always prioritize code readability—use parentheses generously to make complex expressions explicit. Next time, we'll see these operators in heavy use as we learn to control the flow of our programs with conditional statements.