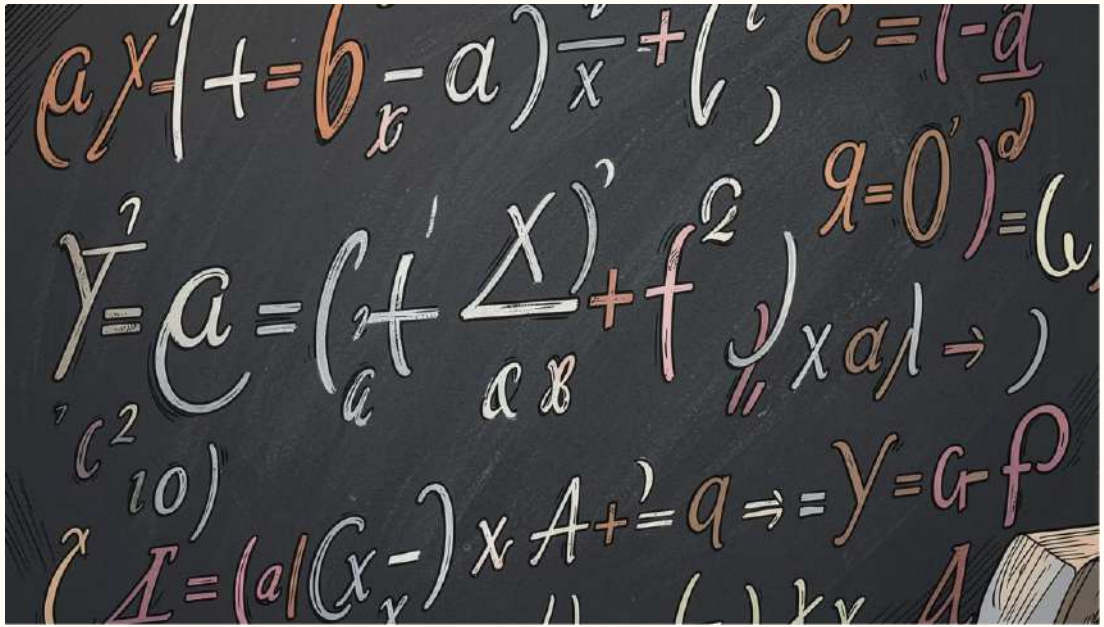
Operators in C Programming

Decoding the symbols that drive logic and computation — from basic arithmetic to advanced bit manipulation.

S. Daniel
Assistant Professor, Department of Mathematics,
Cardamom Planters Association College.

Defining the Landscape



Every C expression is built from two core elements. Understanding their relationship is the first step to writing correct code.

Operands

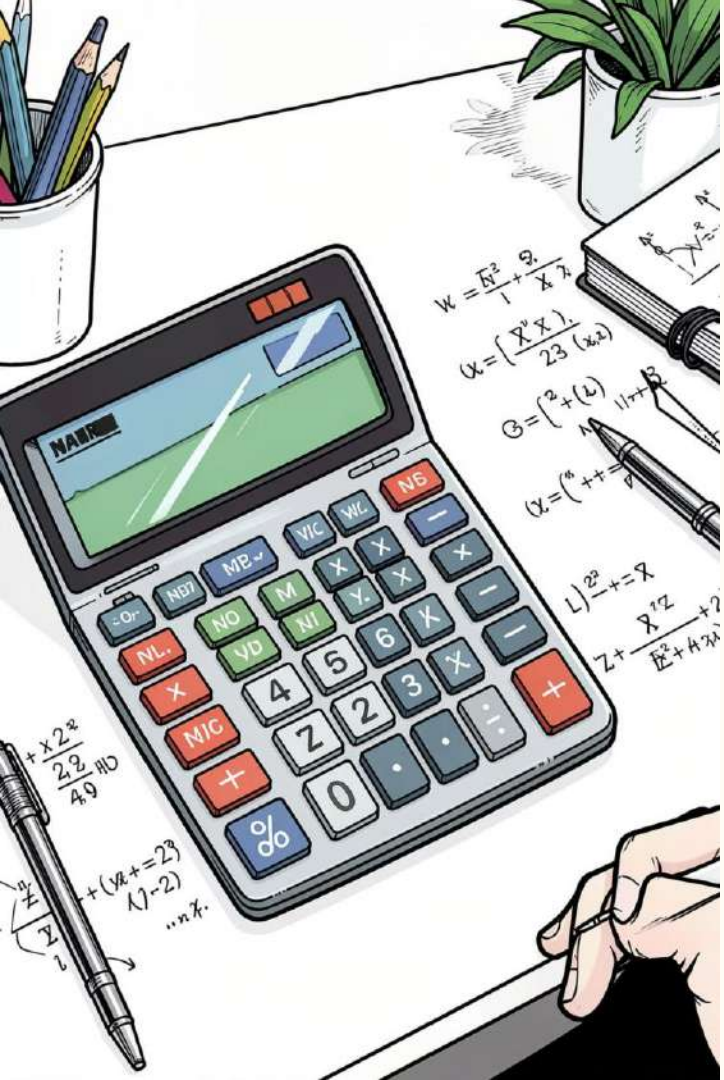
The data items being manipulated — variables, constants, or literals.

Operators

Symbols specifying the action to perform on operands.

Classification

Unary (1 operand), **Binary** (2 operands), **Ternary** (3 operands).



Arithmetic Operators: The Foundation

+ - * /

Standard binary operators for addition, subtraction, multiplication, and division.

% Modulus

Returns the remainder of integer division. Example: `7 % 3 = 1`

Integer Division

Division truncates decimals toward zero. `5 / 2 = 2`, not 2.5

```
int a = 9, b = 4;
printf("%d", a / b); // Output: 2
printf("%d", a % b); // Output: 1
```

Increment & Decrement: Unary Power

Prefix vs Postfix

The position of `++` or `--` changes *when* the operation occurs relative to the value being returned.

```
int x = 5;
int a = ++x;  // x=6, a=6
int b = x++;  // b=6, x=7
```

In isolation, `++x` and `x++` behave identically. The difference matters only within larger expressions.

++x (Prefix)

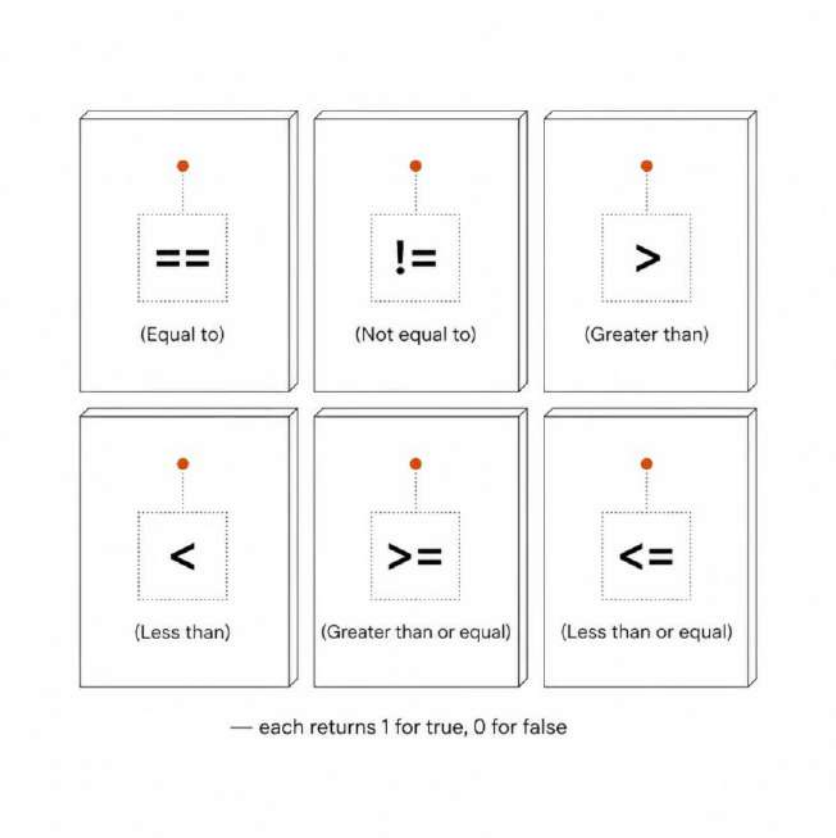
Increments **first**, then returns the new value.

x++ (Postfix)

Returns the current value, **then** increments.

Relational Operators: Logical Decisions

Relational operators compare two values and return `1` (true) or `0` (false), forming the backbone of `if` statements and loops.




```
int x = 10, y = 5;
printf("%d", x == y);  // 0 (false)
printf("%d", x > y);   // 1 (true)
```

⚠ **Critical:** Use `==` for comparison. Using `=` performs assignment and always evaluates as true.

Logical Operators: Complex Conditions

&& (AND)

True only if **both** operands are true. Uses short-circuit evaluation — stops if the first is false.

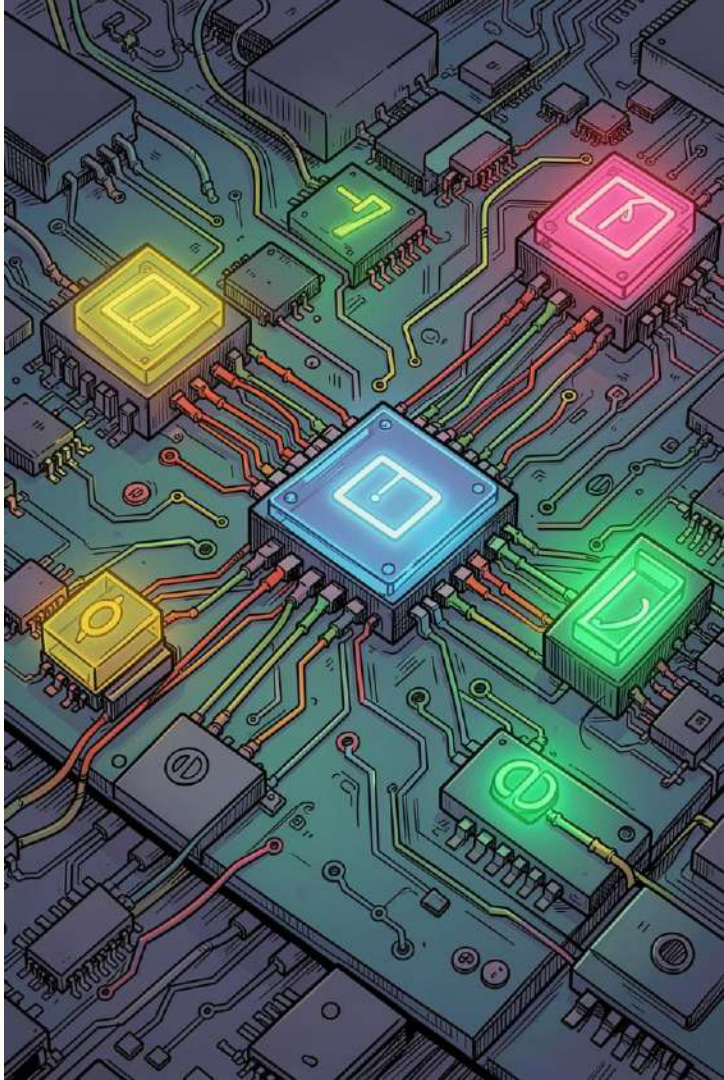
|| (OR)

True if **at least one** operand is true. Stops evaluating once a true is found.

! (NOT)

Inverts the boolean result. `!true` becomes `false`.

```
if (x > 0 && y < 10) { /* safe to proceed */ }
```



Assignment & Compound Operators

Simple Assignment

The `=` operator stores a value in a variable. It is **not** a mathematical equality symbol.

```
int x = 10;
x = x + 5;  // x is now 15
```

Compound Shorthand

C provides shorthand operators that combine arithmetic with assignment, reducing repetition.

01

x += 5

Equivalent to `x = x + 5`

02

x -= 3

Equivalent to `x = x - 3`

03

x *= 2

Equivalent to `x = x * 2`

04

x /= 4

Equivalent to `x = x / 4`

Bitwise & Special Operators

Bitwise Operators

Operate directly on the binary representation of integers — essential for low-level programming and embedded systems.

&

AND

|

OR

^

XOR

~

NOT

<<

Left shift

>>

Right shift

Special Operators

sizeof()

Returns the size of a variable or type in bytes. Example: `sizeof(int)`

? :

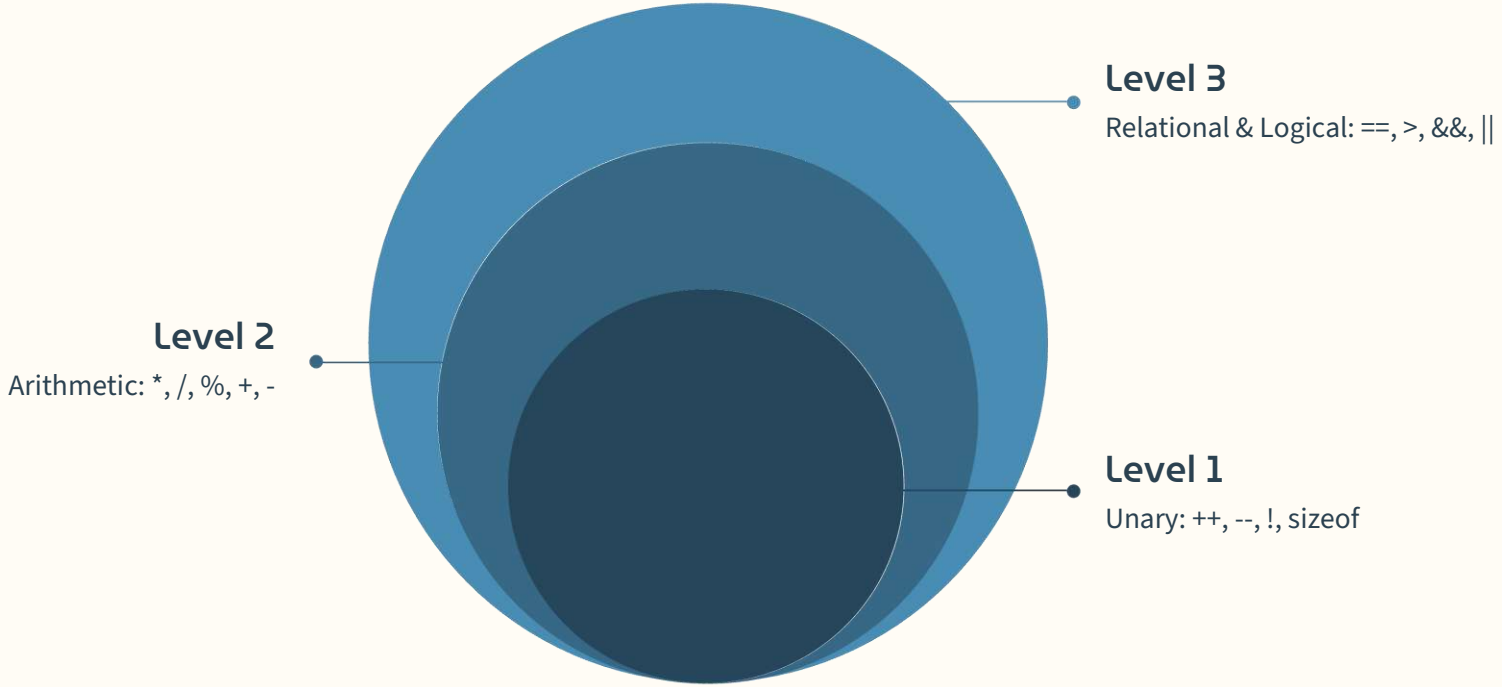
Ternary operator — a compact if-else. `max = (a > b) ? a : b;`

Comma (,)

Evaluates multiple expressions left-to-right; returns the rightmost value.

Precedence & Associativity

C evaluates expressions in a defined order. **Precedence** determines which operator acts first; **associativity** determines the direction when operators share the same precedence level.



Operators at higher levels are evaluated first. Multiplication `*` always executes before addition `+`.

```
int result = a + b * c;
// b * c is evaluated first
// then added to a
```

Rule of thumb: When in doubt, use parentheses `()` to make your intent explicit and avoid ambiguity.

Writing Effective C Code

1 Use parentheses liberally

Clarify intent in complex expressions — readability is never a compromise.

2 Choose the simplest operator

Prefer `x += 5` over `x = x + 5`. Concise code is maintainable code.

3 Master the symbols

Operators are the vocabulary of C. Fluency unlocks efficient, bug-free programming.