Bitwise Operators

Bitwise operators perform operations on the binary forms of numbers.

1.**Bitwise AND (`&`)**: Similar to logical AND.
◦`0 & 0` results in `0`.
◦`0 & 1` results in `0`.
◦`1 & 0` results in `0`.
◦`1 & 1` results in `1`.
◦Example: `4 & 8`. Binary 4 is `100`. Binary 8 is `1000`. To perform the AND operation, you can pad with leading zeros to match the number of bits, making 4 into `0100`.
◦The result is 0 in decimal.

2.**Bitwise OR (|)**: Similar to logical OR.
◦`0 | 0` results in `0`.
◦`0 | 1` results in `1`.
◦`1 | 0` results in `1`.
◦`1 | 1` results in `1`.
◦Example: `4 | 8`. Binary 4 is `0100`. Binary 8 is `1000`.
◦The result is 12 in decimal (1*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 8 + 4 + 0 + 0 = 12).

3.**Bitwise XOR (^)**: Also called Exclusive OR.
◦Rule: If the bits are the same, the result is `0`. If the bits are different, the result is `1`.
◦`0 ^ 0` results in `0`.
◦`1 ^ 1` results in `0`.
◦`0 ^ 1` results in `1`.
◦`1 ^ 0` results in `1`.
◦Example: `4 ^ 8`. Binary 4 is `0100`. Binary 8 is `1000`.
◦The result is 12 in decimal.
◦Example: `3 ^ 7`. Binary 3 is `0011`. Binary 7 is `0111`.
◦The result is 4 in decimal.

4.**Bitwise Left Shift (<<)**: Shifts the binary representation of a number to the left by a specified number of places.
◦Syntax: `n << i`, where `n` is the number and `i` is the number of places to shift.
◦How it works: Each bit in the binary form of `n` is shifted `i` positions to the left. **New places on the right are filled with zeros**. Bits shifted past the leftmost position are lost.
◦Example: `4 << 1`. Binary 4 is `100`. Shift left by 1: `100` becomes `1000`.
◦The result is 8 in decimal.
◦Example: `10 << 2`. Binary 10 is `1010`. Shift left by 2: `1010` becomes `101000`.
◦The result is 40 in decimal (1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 0*2^0 = 32 + 0 + 8 + 0 + 0 + 0 = 40).
◦Relation to multiplication: `a << b` is equivalent to `a * 2^b`. For example, `8 << 1` is `8 * 2^1 = 16`. Binary 8 is `1000`. Shift left by 1: `10000` (16).

5.**Bitwise Right Shift (>>)**: Shifts the binary representation of a number to the right by a specified number of places.
◦Syntax: `n >> i`.
◦How it works: Each bit in the binary form of `n` is shifted `i` positions to the right. **Bits shifted past the rightmost position are discarded**. **New places on the left are filled with zeros** for positive numbers.
◦Example: `10 >> 1`. Binary 10 is `1010`. Shift right by 1: `1010` becomes `0101`. The trailing 0 is discarded.
◦The result is 5 in decimal (0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 0 + 4 + 0 + 1 = 5).
◦Example: `8 >> 2`. Binary 8 is `1000`. Shift right by 2: `1000` becomes `0010`. The trailing 00 are discarded.
◦The result is 2 in decimal. This can be verified in code.
◦Relation to division: `a >> b` is equivalent to `a/2^b`. For example, `8>>1` is `8/2^1 = 4`.
Bitwise operators, especially shift operators, are useful for finding shortcuts in bit manipulation problems, such as checking if a number is a power of two or determining if it's odd or even based on its bits.

Operator Precedence
Operator precedence (or priority) determines the order in which operations are performed in an expression.
•In an expression like `5 - 2 * 6`, multiplication has higher precedence than subtraction, so `2 * 6` is calculated first, then the result is subtracted from 5 (5 - 12 = -7)67.
•A general priority table exists in C++. It's not necessary to memorise the entire table but understand the broad categories.
•General Order of Precedence (from highest to lowest):
◦Unary operators (like `++`, `--`, `!`).
◦Arithmetic operators (`*`, `/`, `%`, `+`, `-`). Multiplication, division, and modulo have equal higher priority than addition and subtraction.
◦Relational operators (`>`, `<`, `>=`, `<=`).
◦Equality operators (`==`, `!=`).
◦Logical AND (`&&`).
◦Logical OR (`||`).
◦Assignment operator (`=`). Assignment has the lowest priority so that the expression on the right is fully calculated before assigning the value to the left.
•Bitwise operators fall somewhere in the middle priority, between arithmetic and logical operators.

•**Parentheses (`()`) can be used to override the default precedence**.
Operations inside parentheses are always performed first. Example: `(5 - 2) * 6` calculates `5 - 2` first (3), then multiplies by 6 (3 * 6 = 18).

•**Associativity** rules are used when operators of equal precedence appear in an expression. Associativity defines the direction in which evaluation proceeds (Left to Right or Right to Left).
◦For `*`, `/`, and `%`, associativity is Left to Right. Example: `4 * 5 % 2`. Left to Right means `4 * 5` is calculated first (20), then `20 % 2` (which is 0). If it were Right to Left, `5 % 2` would be calculated first (1), then `4 * 1` (which is 4).
◦For the assignment operator (=), associativity is Right to Left.
◦For unary operators, associativity is generally Right to Left.
◦Most other operators follow Left to Right associativity.

Scope
Scope defines the region or area in the code where a variable can be accessed or used. Understanding scope is important for managing variable visibility and lifetime. Two main types of scope are discussed:

1.**Local Scope**: Variables declared within a specific block of code are said to have local scope. They are only accessible within that block.

◦Examples of blocks creating local scope:

▪**`if` and `else` blocks**: A variable declared inside an `if` or `else` block is not accessible outside of it. Attempting to access it will result in an error.

▪**Loops (`for`, `while`, etc.)**: Variables declared within the loop's initialisation or inside the loop's body are local to that loop and its iterations. They are not accessible after the loop finishes. This is why the same variable name (like `j`) can be used in separate inner loops without conflict – each `j` is a new variable with its own local scope.

▪**General Block of Code**: Any code enclosed within curly braces `{}` creates a local scope. Variables declared inside these braces are not accessible outside.

▪**Functions**: Variables declared inside a function are local to that function. They exist on the call stack while the function is executing and are destroyed when the function returns. They cannot be accessed from outside the function.

2.**Global Scope**: Variables declared outside of all functions and blocks, usually at the top of the file, have global scope.
◦Global variables are accessible from *anywhere* within the file, including inside different functions.
◦Example: Declaring a variable `x` outside the `main` function makes it a global variable. Both `main` and any other function defined in the same file can access and print the value of `x`.
◦A real-life analogy given is an API key that needs to be accessible by different functions in a program.

Data Type Modifiers
Data type modifiers are used to change the meaning or properties of existing data types, primarily affecting their size and the range of values they can store. They are commonly used with `int`.

1.**`long`**: Used to increase the minimum size of a data type.
◦When used with `int` (`long int`), it typically increases the size from 4 bytes (32 bits) to 8 bytes (64 bits), although this can be system-dependent. `sizeof(int)` might be 4, while `sizeof(long int)` might be 8.
◦A 32-bit integer can store numbers from roughly -2^31 to +2^31 - 1.
◦A 64-bit integer (`long int`) can store numbers from roughly -2^63 to +2^63 - 1, accommodating a much larger range.
◦Can also be used with `double` (`long double`).

2.**`short`**: Used to potentially decrease the size of a data type.
◦When used with `int` (`short int`), it typically decreases the size from 4 bytes to 2 bytes, again, this is system-dependent. `sizeof(short int)` might be 2 bytes.
◦Used when you know the range of values you need to store is small (e.g., storing age, which is unlikely to exceed 150), conserving memory.

3.**`long long`**: Provides even greater capacity than `long`.
◦Used with `int` (`long long int` or simply `long long`.
◦Typically results in an 8-byte capacity, although this can vary by system. `long long` is synonymous with `long long int`.

4.**`signed` and `unsigned`**: Affect whether a data type can store negative values.
◦By default, integer types (`int`, `long int`, `short int`, etc.) are `signed`. This means they can store both positive and negative numbers within their range. One bit (the Most Significant Bit or MSB) is used to represent the sign (0 for positive, 1 for negative).
◦The `unsigned` modifier specifies that the variable will *only* store non-negative (zero or positive) values.
◦When a type is `unsigned`, the MSB is no longer needed for the sign and can be used as part of the number's magnitude.
◦This **doubles the positive range** of the data type. For a 32-bit integer, the range becomes 0 to 2^32 - 1 (compared to 0 to 2^31 - 1 for the positive part of a signed int).
◦`unsigned` is useful for values that are naturally non-negative, like customer IDs or bank account numbers.
◦If you assign a negative value to an `unsigned` variable, it doesn't cause an error, but the value stored and printed will be a large positive number. This is because the compiler interprets the bit pattern (which would represent the negative value in two's complement for a signed type) as a large positive magnitude since there is no sign bit to consider.
Understanding these concepts helps in choosing the appropriate data types and operators for different programming tasks.