Conditional Statements

Conditional Statements are programming constructs used to execute different blocks of code based on whether a condition is true or false.
In real life, conditions determine outcomes, such as needing to be over 18 to vote. In C++, **If-Else statements** are used to implement this logic.
•The **`if`** statement is written in lowercase and is followed by parentheses `()` containing a condition. This condition is a phrase or statement that evaluates to either **true (Yes)** or **false (No)**.
•Following the condition, curly braces `{}` enclose a **block of code** that will execute only if the condition is true. These curly braces are generally recommended for clarity and to group multiple statements, although they are technically optional for a single statement within the `if` block.
•The **`else`** statement follows the `if` block. Its curly braces `{}` enclose a block of code that executes when the `if` condition is false. The `else` statement is optional.

•**Relational operators** (like >=) are used within conditions to compare values, returning true or false. For checking equality, the double equals sign == is used, not a single equals sign. The inequality check uses `!=`.
•Examples of simple If-Else logic include checking if a number is positive or negative, determining voting eligibility based on age, and identifying if a number is odd or even using the modulo operator (`%`). An odd number is one where `n % 2` is not equal to 0 (`n % 2 != 0`).

For scenarios involving **multiple conditions**, the **`else if`** statement is used.
•The first condition is checked with an `if` statement.
•Subsequent conditions are checked using `else if` statements. You can have multiple `else if` statements.
•An optional final `else` statement can be included to handle cases where none of the preceding `if` or `else if` conditions are true.
•This structure allows checking conditions sequentially until one is found to be true, executing the corresponding code block, and then skipping the rest.
•An example of using `else if` is a grading system based on marks. To check if marks fall within a range (e.g., between 80 and 90), the **logical AND operator (`&&`)** is used in the condition (e.g., `marks >= 80 && marks < 90`).

Character case detection provides another example of conditional logic.
•One method is to compare the character directly to the range of lowercase ('a' to 'z') or uppercase ('A' to 'Z') characters using >= and <= along with `&&`. For example, checking for lowercase is `ch >= 'a' && ch <= 'z'`.
•An alternative method uses the **ASCII values** of characters. Each character has a corresponding numerical value. Capital letters 'A' through 'Z' have ASCII values from 65 to 90, and lowercase letters 'a' through 'z' have values from 97 onwards.
•By comparing a character's ASCII value to these numerical ranges, you can determine its case. For example, `ch >= 65 && ch <= 90` checks for uppercase.
•When comparing a character to a number, the compiler performs **implicit type conversion**, converting the character to its ASCII integer value for the comparison.

A simplified way to write basic If-Else logic is the **Ternary Statement**.
•It has a three-part structure: `condition ? statement_if_true : statement_if_false`.
•The condition is followed by a question mark `?`.
•The statement to execute if the condition is true comes next, followed by a colon `:`.
•The statement to execute if the condition is false comes after the colon.
•Ternary statements are generally used for very simple conditions and are not preferred for complex logic or multiple statements within the true/false branches, as they can be harder to read.

Loops are fundamental programming constructs used to repeat a block of code multiple times. This is useful for tasks that require repetition, like printing numbers from 1 to 500. There are three main types of loops in C++: **While loops, For loops, and Do-While loops**.
•The **For loop** is the most frequently used loop in programming, especially for DSA (Data Structures and Algorithms).
•The **While loop** is also used often.
•The **Do-While loop** is rarely used.
•Any task achievable with one type of loop can generally be done with the others; the choice often depends on syntax preference or specific requirements.

The **While Loop** structure involves a condition that is checked *before* each iteration.
•Syntax: `while (condition) { code_to_repeat }`.
•The code block inside the curly braces repeats as long as the condition remains true.
•To prevent an **infinite loop** (a loop that never stops because its condition never becomes false), you typically need an initial variable (a counter), a condition that depends on this variable, and a step inside the loop that **updates** the variable to eventually make the condition false.
•Updating the variable can be done using `count = count + 1`, `count += 1`, or the more common **increment operator ++** (`count++`).
•An example is printing numbers from 1 to n. A counter variable `i` starts at 1, the condition is `i <= n`, and `i` is incremented (`i++`) inside the loop. If the update step is missing, the loop condition might always remain true, resulting in an infinite loop.

The **For Loop** is a more concise way to write loops, combining the initialization, condition, and update steps into a single line.
•Syntax: `for (initialization; condition; update) { code_to_repeat }`.
•The `initialization` part runs once before the loop starts (e.g., `int i = 1;`).
•The `condition` is checked before each iteration (e.g., `i <= n;`).
•The `update` part runs after each iteration of the loop (e.g., `i++;`).
•The code block inside the curly braces `{}` executes after the condition is checked and is true.
•The `initialization` statement is executed only once, whereas the `condition`, the code block, and the `update` statement repeat.
•Loop counter variables are often named simply `i`, `j`, or `k`, which is an exception to the general convention of using meaningful variable names written in **camel case** (like `totalSum`).

•An example is calculating the sum of numbers from 1 to n. An outer variable `sum` is initialized to 022. Inside a For loop running from `i = 1` to `n`, the current value of `i` is added to `sum` (`sum += i`).

The **`break` statement** is a keyword that can be used inside a loop to terminate it immediately, regardless of the loop's condition.
•It's useful when a condition is met that makes further loop execution unnecessary.
•For example, in the sum calculation, you could use `break` to stop the loop early if the counter variable reaches a certain value, even if the loop was intended to run longer.
•**Keywords** like `break`, `int`, and `while` are special reserved words in C++ with predefined meanings and cannot be used as variable names.

Loops and conditional statements can be combined to solve more complex problems, such as finding the sum of only the **odd numbers** within a range from 1 to n.
•A loop iterates through numbers from 1 to n.
•Inside the loop, an `if` condition checks if the current number is odd (`i % 2 != 0`).
•If the condition is true (the number is odd), it is added to a separate sum variable dedicated to odd numbers.

The **Do-While Loop** is similar to the While loop but with a key difference in execution order.
•Syntax: `do { code_to_repeat } while (condition);`. Note the semicolon after the `while` condition.
•The code block inside the `do` part executes **at least once** before the `while` condition is checked.
•If the condition is false initially, the code block still runs one time. In a standard While loop, the code block wouldn't run at all if the condition is false from the start.
•Do-While loops are less commonly used than For and While loops.

Nested Loops involve placing one loop inside the body of another loop.
•The outer loop controls the overall structure, often the number of rows in a pattern.
•The inner loop executes completely for each single iteration of the outer loop, often controlling what is printed within a row (e.g., the number of columns or items).
•It is best practice to use different variable names for the outer and inner loop counters (e.g., `i` for the outer loop and `j` for the inner loop).
•Nested loops are particularly useful for generating patterns, such as printing a rectangle of stars. The outer loop runs for the desired number of lines, and the inner loop runs for the desired number of stars per line.