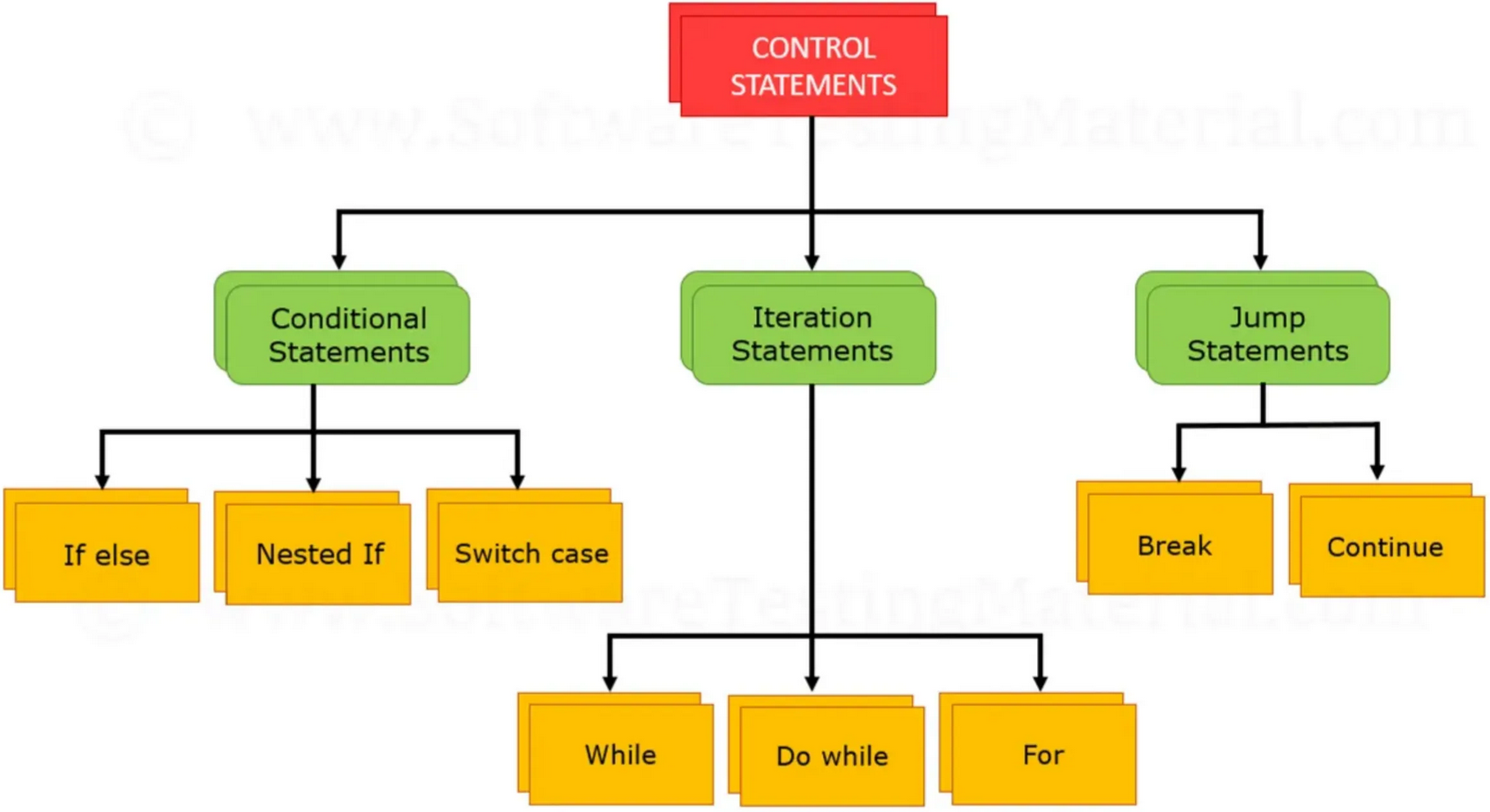
****

Control statements are fundamental components in programming languages that **determine the flow of execution** in a program. Rather than executing code line by line in a straight sequence, control statements enable the program to make **logical decisions**, **repeat specific blocks of code**, or **divert the flow** to different parts of the code based on certain conditions.

They allow the program to:

* **Make decisions** using conditional statements like **if, if-else, and switch.** These statements check for specific conditions and execute code blocks accordingly.
* **Repeat operations** through loops like **for, while, and do-while**, which are essential for tasks that require repetition, such as processing lists, computing sums, or reading input until a condition is met.
* **Jump between sections of code** using jump statements like **break, continue, and goto,** which help manage how and when loops or conditional structures are exited or skipped.

In essence, control statements are what give **intelligence** and **dynamism** to a program, allowing it to act based on data, user input, or outcomes of previous operations, making programs more flexible and powerful.

## **1. Conditional Statements**

Conditional statements are used in programming to make decisions based on whether a condition is true or false. They allow a program to **execute different blocks of code** depending on certain logical conditions. The most common conditional statements include if, if-else, else-if, nested if, and switch.

### 🔸 a. If else

The if-else statement is one of the most basic and commonly used conditional statements in programming. It allows the program to **choose between two paths** based on whether a given condition is true or false.

* + If the condition is **true**, the code inside the if block is executed.
  + If the condition is **false**, the code inside the else block is executed instead.
* Used when you want to check a condition and do **one of two** things:

✅ Simple Example:

int age = 20;

if (age >= 18) {

printf("You are an adult.");

} else {

printf("You are a minor.");

}

### 🔸 b. Nested if

A **nested if** is an if statement placed **inside another if or** else **block**. It allows a program to check **multiple conditions in a structured way**, making decisions that depend on **more than one level of logic**.

This is useful when one condition must be **true before checking** another condition.

* When an if is placed **inside another** if, it’s called a nested if.

✅ Simple Example:

int marks = 85;

if (marks >= 60) {

if (marks >= 80) {

printf("Excellent!");

} else { printf("Good job!"); }

} else { printf("You need to work harder.");}

### 🔸 c. Switch case

The switch statement is a control structure used to **select one of many possible blocks of code** to be executed, based on the **value of a variable or expression**. It’s an alternative to using multiple if-else-if statements when checking for **equality** against fixed values (like numbers or characters).

It makes code **cleaner and easier to read**, especially when dealing with a large number of specific options.

* Best for checking **equality** against multiple constant values (integers or characters).

✅ Simple Example:

int day = 3;

switch(day) {

case 1:

printf("Monday");

break;

case 2:

printf("Tuesday");

break;

case 3:

printf("Wednesday");

break;

default:

printf("Invalid day");

}

## 2. Iteration Statements (Loops)

Iteration statements, commonly known as **loops**, are used in programming to **repeat a block of code** multiple times until a specific condition is met. Instead of writing the same code again and again, loops allow you to automate repetition, making programs more efficient and concise. There are mainly three types of loops: **for, while, and do-while.** The **for** loop is typically used when the number of iterations is known in advance, while the **while** and **do-while** loops are used when the number of repetitions depends on a condition being true. Loops are essential for tasks such as processing arrays, handling user input, generating sequences, and performing repeated calculations.

### 🔸 a. While

* The **condition is checked first**, then the loop runs.

✅ Simple Example:

int i = 1;

while (i <= 5) {

printf("%d\n", i);

i++;

}

In C programming, a while loop with the condition 1 (i.e., while (1)) creates an infinite loop. This means the loop will execute continuously without stopping unless a break statement is used or the program is externally interrupted. The condition 1 is always true because the value 1 is considered a non-zero value, which evaluates to true in C.

#include <stdio.h>

int main() {

while (1) {

printf("This will run forever!\n");

// You can break the loop with a condition, for example:

// if (some\_condition) {

// break;

// }

}

return 0; }

This kind of loop is typically used in programs that require continuous execution, such as in embedded systems or event-driven programs, where the loop keeps checking for conditions or input until the program needs to exit.

### 🔸 b. Do while

A do-while loop in C is similar to a while loop, but with one key difference: the condition is checked after the loop body is executed, which means the code inside the loop will always execute at least once, even if the condition is initially false.

* The loop **runs once before checking the condition**.

✅ Simple Example:

int i = 1;

do {

printf("%d\n", i);

i++;

} while (i <= 5);

**Used When the Code Should Run At Least Once**: This is useful when you want to perform an action first, and then check whether to continue based on the result of that action.

### 🔸 c. For

In C programming, a for loop is a control flow statement used to repeat a block of code a specific number of times. It consists of three parts: initialization, condition, and increment/decrement, all written in a single line, making it compact and easy to manage. The syntax is: for(initialization; condition; update), where initialization sets the starting point, the condition is checked before each iteration, and the update changes the loop variable after each iteration.

* Used when the number of iterations is **known**.

✅ Simple Example:

for (int i = 0; i < 5; i++) {

printf("i = %d\n", i);

}

## 3. Jump Statements

In C programming, jump statements are used to transfer control from one part of the program to another, breaking the normal sequential flow. The main jump statements are break, continue, goto, and return. The break statement is used to exit loops or switch statements prematurely, while continue skips the current iteration of a loop and jumps to the next one. The goto statement allows jumping to a labeled statement within the same function, though it's generally discouraged due to potential readability issues. The return statement is used to exit a function and optionally return a value to the calling function. These statements are powerful but should be used carefully to maintain code clarity and structure.

### 🔸 a. Break

* Used to **exit** from a loop or switch early.

✅ Simple Example:

for (int i = 1; i <= 10; i++) {

if (i == 5) {

break;

}

printf("%d\n", i);

}

### 🔸 b. Continue

* Skips the current iteration and **moves to the next** one.

✅ Simple Example:

for (int i = 1; i <= 5; i++) {

if (i == 3) {

continue;

}

printf("%d\n", i);

}

**🟡 Here,** 3 **will be skipped.**

### 🔸 c. goto

* The goto statement is used to **jump to another part of the program** labeled with a label:. It allows **unconditional jumps**, meaning it **bypasses the normal flow** of execution.

## ⚠️ When to use goto?

* Mostly used to **break out of deeply nested loops**.
* In modern programming, it's **discouraged** because it can make code **hard to read and debug**.
* But still useful in **low-level programming**, like embedded systems or state machines.

goto label\_name;

// ... later in the code

label\_name:

// code to execute

✅ Simple Example:

#include <stdio.h>

int main() {

int num = 5;

if (num == 5) {

goto skip;

}

printf("This won't print if num == 5\n");

skip:

printf("Jumped to the label!\n");

return 0;

}