**Decision Control Structures**

1. **If Statement**  
   The if statement is used to execute a block of code if a specified condition is true.  
   **Syntax**:

c

if (condition) {

// Code to execute if condition is true

}

**Example**:

int a = 5;

if (a > 0) {

printf("a is positive");

}

**Explanation**: If a > 0, it prints "a is positive."

1. **If-else Statement**  
   The if-else statement adds an alternative block of code if the condition is false.  
   **Syntax**:

if (condition) {

// Code if condition is true

} else {

// Code if condition is false

}

**Example**:

int a = -5;

if (a > 0) {

printf("a is positive");

} else {

printf("a is negative");

}

**Explanation**: If a is not greater than 0, it prints "a is negative."

1. **Nested If**  
   The nested if is used when one if statement is inside another if statement.  
   **Syntax**:

if (condition1) {

if (condition2) {

// Code if both conditions are true

}

}

**Example**:

int a = 5, b = 3;

if (a > 0) {

if (b > 0) {

printf("Both are positive");

}

}

1. **If-else Ladder**  
   This is a chain of if-else statements used when multiple conditions are checked in sequence.  
   **Syntax**:

if (condition1) {

// Code for condition1

} else if (condition2) {

// Code for condition2

} else {

// Code if no condition is true

}

**Example**:

int marks = 85;

if (marks >= 90) {

printf("Grade A");

} else if (marks >= 75) {

printf("Grade B");

} else {

printf("Grade C");

}

1. **Switch Case Statement**  
   The switch statement allows choosing between multiple cases based on the value of a variable.  
   **Syntax**:

switch (variable) {

case value1:

// Code for value1

break;

case value2:

// Code for value2

break;

default:

// Code if none of the values match

}

**Example**:

int day = 2;

switch (day) {

case 1:

printf("Monday");

break;

case 2:

printf("Tuesday");

break;

default:

printf("Other day");

}

**Explanation:** The switch case for 2 is executed, so "Tuesday" is printed.

**Iterative Statements**

1. **For Loop**  
   The for loop repeats a block of code a specific number of time

**Syntax**:

for (initialization; condition; increment) {

// Code to be repeated

}

**Example**:

int I;

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

printf("%d ", i);

}

// Output: 0 1 2 3 4

1. **While Loop**

A While loop repeats a block of code as long as a condition is true  
**Syntax**:

while (condition) {

// Code to be repeated

}

**Example**:

int i = 0;

while (i < 5) {

printf("%d ", i);

i++;

}

// Output: 0 1 2 3 4

1. **Do-while Loop**

A do-while loop runs the block of code at least once, then checks the condition.  
**Syntax**:

do {

// Code to be repeated

} while (condition);

**Example**:

int i = 0;

do {

printf("%d ", i);

i++;

} while (i < 5); //Output: 0 1 2 3 4

**Conditional Statements**

* **Break**: Exits the loop or switch case immediately.
  + **Example**:

Int i;

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

if (i == 3) break; // Loop exits when i is 3

printf("%d ", i);

}

// Output: 0 1 2

* **Continue**: Skips the current iteration and moves to the next iteration of the loop.
  + **Example**:

c

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

if (i == 2) continue; // Skips when i is 2

printf("%d ", i);

}

// Output: 0 1 3 4

**Storage Classes in C**

Storage classes in C define the **scope**, **visibility**, and **lifetime** of variables/functions within a C program. The four main storage classes are:

**1. Automatic (auto):**

**- Default** for local variables.

**-** Stored in the **stack.**

**-** Lifetime is within the function/block.

**-** Not visible outside the block (local scope).

**2. External (extern):**

**-** Used to declare a global variable or function that is defined in another file or later in the code.

**-** Stored in **global memory.**

**-** Scope**: Global** (accessible across files).

**3. Static:**

**-** Retains the value of a variable even after the scope in which it was declared has existed.

- For local variables, retains the value between function calls.

- For global variables, restricts the variable's scope to the file in which it is declared.

**4. Register:**

**-** Suggests that the variable be stored in a **CPU register** for fast access.

**-** Scope:Same as local variables, but faster access.

**-** Limited size depending on the processor's registers.

**---**

**Arrays in C**

An **array** is a collection of data items of the **same type** stored at contiguous memory locations. Arrays allow for **efficient access** to multiple values using an index.

**Declaration of an Array**

**```**

data\_type array\_name[size];

**```**

**- `data\_type`:** The type of elements the array will hold (e.g., `int`, `float`).

**- `array\_name`:** The identifier for the array.

**- `size`:** Number of elements the array can hold.

**Example:**

**```**

int numbers[10]; // Array to hold 10 integers**.**

**```**

**Initialization of an Array**

Arrays can be initialized at the time of declaration:

**```**

int numbers[5] = {1, 2, 3, 4, 5}; // Initializes all elements.

**```**

**Or partially:**

**```**

int numbers[5] = {1, 2}; // Remaining elements are initialized to 0.

**```**

**Types of Arrays**

**1.** Single Dimensional Array:

- A linear list of elements.

- Accessed using a single index.

**- Example:**

**```**

int arr[5] = {10, 20, 30, 40, 50};

printf("%d", arr[2]); // Output: 30

**```**

**2. Two Dimensional Array:**

**-** Represents a matrix or a table of elements.

- Accessed using two indices: `arr[row][column]`.

**- Example:**

**```**

int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

printf("%d", matrix[1][2]); // Output: 6

**```**

**---**

Address Calculation of an Element in a 2-D Array

For a **2-D array** declared as:

**```**

int arr[rows][columns];

**```**

The **address** of an element at position `arr[i][j]` (i-th row, j-th column) can be calculated as:

**Address = Base Address + [(i × total\_columns) + j] × size\_of\_data\_type**

Where:

- `Base Address` is the starting memory location of the array.

- `i` is the row index, `j` is the column index.

- `total\_columns` is the number of columns in the array.

- `size\_of\_data\_type` is the size (in bytes) of the array's data type.

This formula helps map the 2D array to contiguous memory locations.