Fundamentals of Programming Languages

Unit III: Control Flow

(06 Hours)

Topics Covered:

Decision Making and Branching

- Simple If Statement
- If-Else Statement
- Else-If Ladder
- Switch Statement
- Goto Statement

Decision Making and Looping

- While Statement
- Do-While Statement
- For Statement
- Break Statement
- Continue Statement

C Programming Language

Course Overview: Control Flow in C

Topics covered:

Decision Making

- Simple If Statement
- If-Else Statement
- Else-If Ladder
- Switch Statement
- Goto Statement

Looping

- While Statement
- Do-While Statement
- For Statement
- Break Statement
- Continue Statement

Each control flow statement will be explored with syntax explanations, practical code examples, and common use cases to build strong programming fundamentals.

Introduction to Control Flow

Control flow allows programmers to dictate the order in which instructions are executed. It's fundamental to creating dynamic, responsive programs.

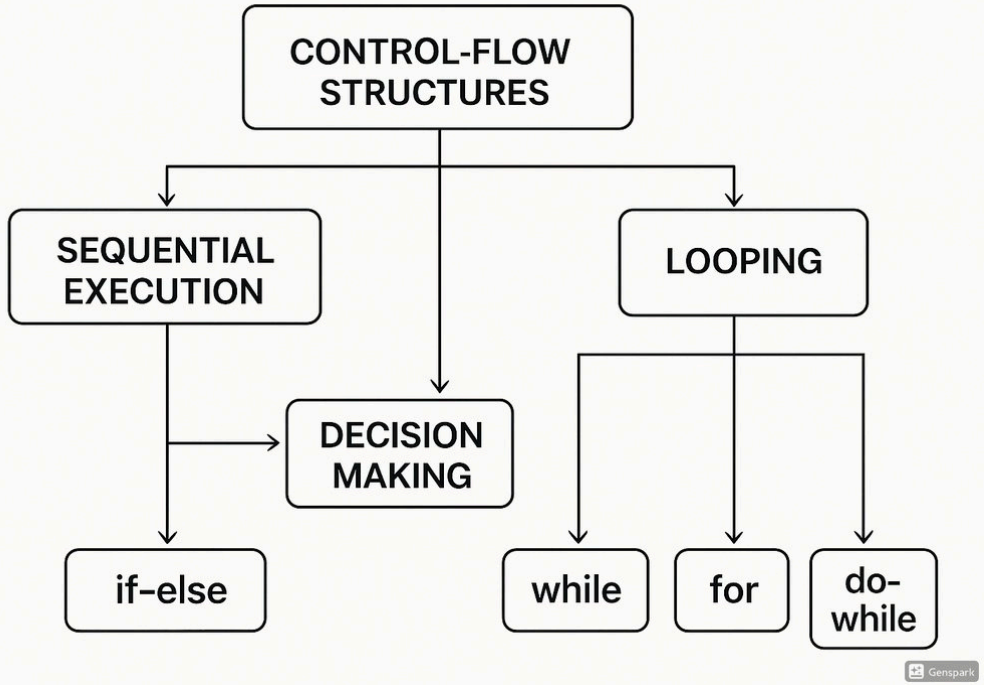
Three Main Control Structures in C:

1. **Sequential**
   Default execution from top to bottom, one statement after another.

2. **Selection (Decision Making)**
   Executing different code blocks based on conditions: if, if-else, switch.

3. **Iteration (Looping)**
   Repeatedly executing code blocks: while, do-while, for loops.

Understanding control flow is essential for writing structured, efficient programs.



Control Flow Structure Visualization

Control Flow Structures – Visual Overview

Control flow structures determine the execution path of a program. The flowchart illustrates how these structures interact and direct program flow.

Key Control Structures Illustrated:

Sequential Execution

Program statements execute in order from top to bottom

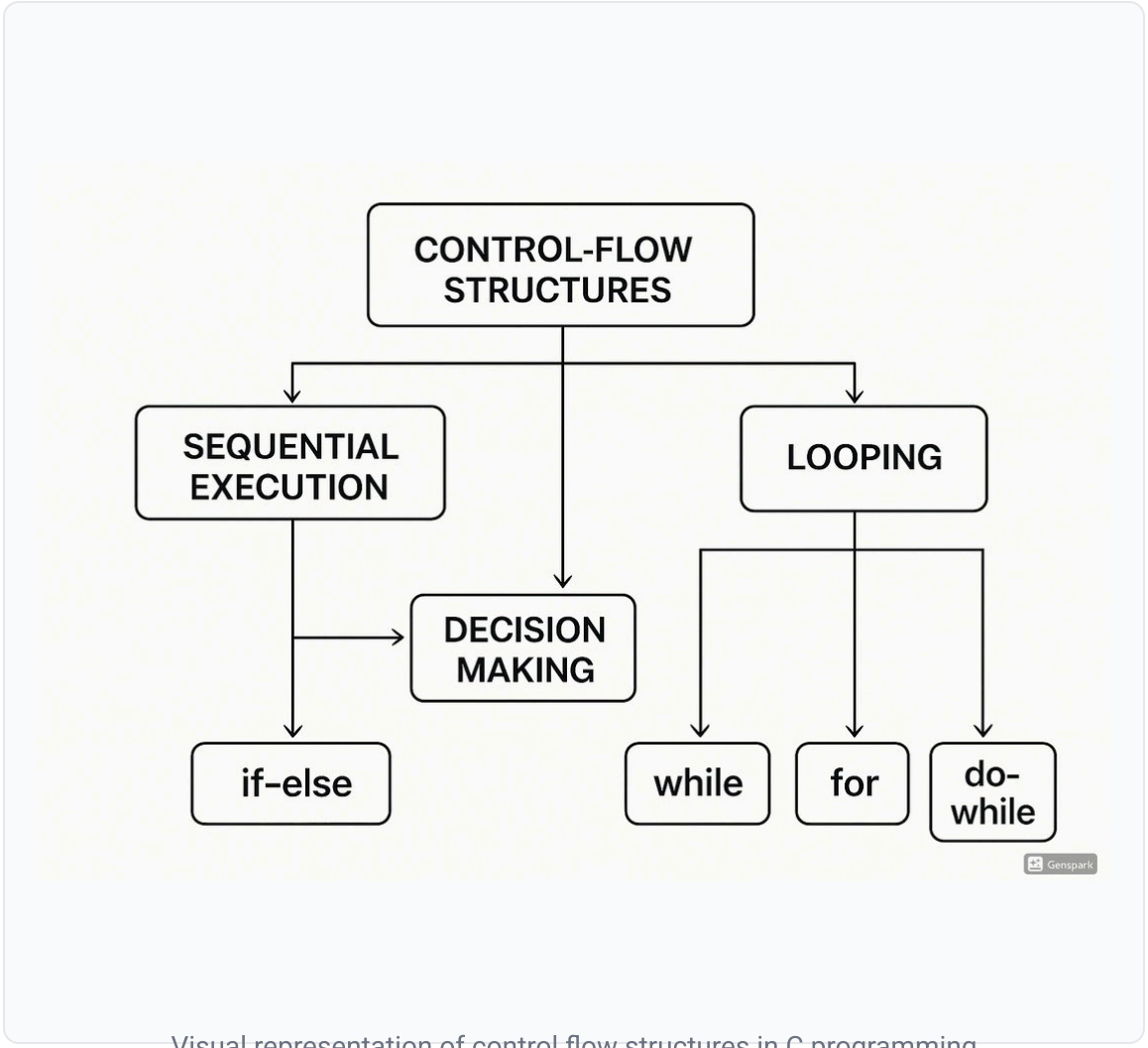
Decision Making (Branching)

Conditional execution based on true/false evaluations

Loops (Iteration)

Repeated execution of code blocks until a condition changes

The diagram shows how these structures can be combined to create complex program behaviors.



Visual representation of control flow structures in C programming

Decision Making Concepts

Conditional Statements in C

Core Decision Statements

- **Simple If:** Executes code only when condition is true
- **If-Else:** Provides alternative execution path when condition is false
- **Else-If Ladder:** Tests multiple conditions sequentially
- **Switch-Case:** Multi-way branching based on expression value

Key Concepts

- Conditions evaluate to true (non-zero) or false (zero)
- Conditional statements allow program to make choices
- Selection of code paths happens at runtime
- Boolean operators can create complex conditions
- Proper indentation improves code readability

Decision making statements allow C programs to choose different code paths based on conditions. They form the foundation for implementing program logic and creating responsive software that can adapt to different inputs and scenarios.

Simple if Statement in C

Syntax

```
if (condition) {
    // code to be executed if
    // condition is true
    statement1;
    statement2;
    ...
}
```

How it works:

- The `if` statement evaluates a condition inside parentheses `()`
- If the condition evaluates to `true` (non-zero), the code block inside the curly braces `{}` is executed
- If the condition is `false` (zero), the code block is skipped
- For a single statement, curly braces `{}` are optional but recommended

Example

```c
#include <stdio.h>

int main() {
    // Number of people in the audience
    int num = 100;

    // Conditional code inside if statement
    if (num > 50) {
        printf("Start the show\n");
    }

    return 0;
}
```

Output:

```
Start the show
```

Key Points

- ✓ Common conditions use comparison operators: `==`, `!=`, `>`, `<`, `>=`, `<=`
- ✓ Logical operators can combine conditions: && (AND), || (OR), ! (NOT)

If-Else Statement in C

Syntax

```
if (condition) {
    // code executed when condition is true
    statement1;
    statement2;
    ...
} else {
    // code executed when condition is false
    statement3;
    statement4;
    ...
}
```

How it works:

- The `if` statement evaluates a condition inside the parentheses
- If the condition evaluates to `true`, the first block of code is executed
- If the condition is `false`, the code block after `else` is executed
- The `else` block is optional and provides an alternative execution path

Example

```
#include <stdio.h>

int main() {
    int i = 10;

    // If-else statement
    if (i > 18) {
        printf("Eligible for vote\n");
    } else {
        printf("Not Eligible for vote\n");
    }

    return 0;
}
```

Output:

```
Not Eligible for vote
```

Key Points

- Only one of the two blocks (if or else) will be executed, never both
- The `else` statement cannot exist without a preceding `if`
- Used when a decision needs exactly two alternative actions

Else-If Ladder in C

Syntax for Multi-way Branching

```
if (condition1) {
    // executed if condition1 is true
    statement1;
}
else if (condition2) {
    // executed if condition1 is false
    // and condition2 is true
    statement2;
}
else if (condition3) {
    // executed if condition1 and condition2
    // are false and condition3 is true
    statement3;
}
else {
    // executed if all conditions are false
    defaultStatement;
}
```

How it works:

- Conditions are evaluated from top to bottom
- Once a condition is `true`, its block is executed and the rest are skipped
- The `else` block executes only if all conditions are `false`
- You can have multiple `else if` statements (unlimited)

Example

```
#include <stdio.h>

int main() {
    int i = 20;

    // If-else-if ladder with three conditions
    if (i == 10)
        printf("Not Eligible");
    else if (i == 15)
        printf("Wait for three years");
    else if (i == 20)
        printf("You can vote");
    else
        printf("Not a valid age");

    return 0;
}
```

Output:

```
You can vote
```

Key Points

- Similar to `switch` statement but can evaluate any type of condition
- Better than nested if statements for sequential condition checking
- Useful for implementing different actions based on multiple possible conditions

Switch Statement in C

Syntax

```
switch (expression) {
    case value1:
        // code to be executed if
        // expression equals value1
        statement(s);
        break;
    case value2:
        statement(s);
        break;
    // more cases as needed
    default:
        // code to be executed if
        // expression doesn't match any case
        statement(s);
        break;
}
```

How it works:

- The `switch` evaluates an expression (must be integer or character)
- Compares the result with the values specified in `case` statements
- When a match is found, the statements after that `case` execute
- The `break` statement terminates the switch block (prevents fall-through)
- The `default` case executes if no match is found

Example

```
#include <stdio.h>

int main() {
    // variable to be used in switch statement
    int var = 18;

    // declaring switch cases
    switch (var) {
    case 15:
        printf("You are a kid\n");
        break;
    case 18:
        printf("Eligible for vote\n");
        break;
    default:
        printf("Default Case is executed\n");
        break;
    }

    return 0;
}
```

Output:

```
Eligible for vote
```

Key Points

- The `switch` expression must evaluate to an integer or character
- Without `break`, execution continues to the next case (called "fall-through")
- The `default` case is optional, but good practice to include
- Use `switch` when testing a variable against multiple values

Goto Statement in C

Syntax

```
// Define a label
label_name:

// Jump to the label
goto label_name;
```

How it works:

- The `goto` statement provides an unconditional jump to a labeled statement within the same function
- A `label` is an identifier followed by a colon `:`
- When program execution reaches a `goto` statement, control immediately transfers to the labeled statement
- Labels have function scope (visible only within the function they are defined)

Example

```
#include <stdio.h>

int main() {
    int n = 1;

    // Define a label
label:
    printf("%d ", n);
    n++;

    // If n is less than or equal to 10,
    // jump back to the label
    if (n <= 10)
        goto label;

    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

Key Points

- ⚠ Use `goto` sparingly as it can make code harder to read and maintain
- ✓ Useful for breaking out of deeply nested loops or error handling
- ✓ Cannot jump between functions or skip variable initializations

Introduction to Looping Concepts

Loops in C Programming:

What are Loops?

- Loops allow repetition of code blocks until a condition is met
- Used to automate repetitive tasks without code duplication
- Essential for traversing arrays and data structures
- Enable iterative calculations and algorithms
- Provide controlled program execution flow

Types of Loops in C

- **while** - Condition checked before loop body executes
- **do-while** - Condition checked after loop body executes
- **for** - Compact syntax with initialization, condition, and update
- **break** - Exit loop early under certain conditions
- **continue** - Skip current iteration, proceed to next

Proper loop selection and implementation is crucial for program efficiency. The right loop type depends on whether you need pre-test vs. post-test behavior, known iteration count, or specific termination conditions. Each loop structure provides unique advantages for different programming scenarios.

While Loop in C

Syntax

```
while (condition) {
    // code to be executed
    // as long as condition is true
    statement1;
    statement2;
    ...
    // update statement (important)
    update_condition;
}
```

How it works:

- The `while` loop evaluates a condition before each iteration
- If the condition is `true`, the code inside the loop executes
- After execution, control returns to evaluate the condition again
- The loop continues until the condition becomes `false`

⚠ Without an update statement, you risk creating an **infinite loop**

Example

```
#include <stdio.h>

int main() {
    // Initialize counter variable
    int i = 1;

    // while loop from 1 to 5
    while (i <= 5) {
        printf("%d ", i);

        // Update counter (critical)
        i++;
    }

    return 0;
}
```

Output:

```
1 2 3 4 5
```

Key Points

- ✓ The `while` loop is entry-controlled (condition checked before iteration)
- ✓ If condition is initially false, the loop body never executes
- ✓ Common uses: reading input until a condition, array traversal, waiting for events

Do-While Loop in C

Syntax

```
do {
    // code to be executed
    statement1;
    statement2;
    ...
} while (condition);
```

How it works:

- The code block inside `{}` is executed at least once before condition is checked
- After execution, the `condition` inside parentheses is evaluated
- If condition is `true`, the loop continues and code executes again
- If condition is `false`, the loop terminates
- Notice the semicolon `;` after the while condition - it's required

Example

```
#include <stdio.h>

int main() {
    int i = 1;

    // do-while loop execution
    do {
        printf("%d ", i);
        i++;
    } while (i <= 5);

    return 0;
}
```

Output:

```
1 2 3 4 5
```

Key Points

- Unlike `while`, the `do-while` loop guarantees at least one execution
- Use when code needs to run before checking the condition
- Common uses: menu systems, input validation, processes requiring at least one execution

For Loop in C

Syntax

```
for (initialization; condition; increment) {
    // code to be executed in each
    // iteration of the loop
    statement1;
    statement2;
    ...
}
```

How it works:

- `initialization`: Executes once at the beginning (typically initializes a counter variable)
- `condition`: Evaluated before each loop iteration; loop continues if true, exits if false
- `increment`: Executes after each iteration (typically modifies the counter variable)
- All three expressions are optional, but semicolons are required

Example

```
#include <stdio.h>

int main() {
    // Print numbers from 1 to 5
    for (int i = 1; i <= 5; i++) {
        printf("%d ", i);
    }

    return 0;
}
```

Output:

```
1 2 3 4 5
```

Key Points

- ✓ Ideal for known number of iterations (e.g., array traversal)
- ✓ Counter variable can be modified inside the loop body
- ✓ Multiple initializations/increments using commas: `for(i=0,j=0; i<5; i++,j+=2)`

Break and Continue in C

Break Statement

```
break;
```

- Terminates the loop or switch statement
- Control flows to the statement immediately after the loop/switch

Continue Statement

```
continue;
```

- Skips the current iteration of a loop
- Control jumps to the loop's update statement (for loop) or condition check (while/do-while)

Examples

Break Example:

```c
#include <stdio.h>

int main() {
    int arr[] = { 1, 2, 3, 4, 5 };
    int key = 3, i;

    for (i = 0; i < 5; i++) {
        if (arr[i] == key) {
            printf("Element found at index: %d\n", i);
            break; // Exit loop when found
        }
    }

    printf("Search completed\n");
    return 0;
}
```

Continue Example:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 6) {
            continue; // Skip 6
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output (Break Example):

```
Element found at index: 2
Search completed
```

Output (Continue Example):

```
1 2 3 4 5 7 8 9 10
```

Common Use Cases

- `break`: Exiting loops early when a condition is met
- `continue`: Skipping specific iterations based on conditions

Summary & Conclusion

Key Takeaways

Decision Making

- If statements evaluate conditions to execute code selectively
- Switch provides efficient multi-way branching for constant expressions
- Use goto sparingly for specific control flow situations

Looping

- While loops check condition before execution
- Do-while ensures at least one iteration by checking after execution
- For loops offer compact syntax for iteration with counter variables
- Break and continue provide fine-grained control within loops

Practical Applications

Mastery of control flow statements is foundational for developing:

- Data processing algorithms
- Search and sort operations
- User input validation
- System control functions
- Game development logic
- Automation systems

→ Continue practicing with real-world examples to solidify your understanding of control flow structures in C programming.