Control Statements (Looping)

C Programming Journey

Introduction

Computers excel at performing repetitive tasks swiftly, loops offer efficiency, especially when executing code numerous times. When parts of our code need to execute multiple times, we can either repeat the code or use loops. The process involves repeating a statement or group of statements until a termination condition is met. The C programming language provides various loop commands for this purpose. Loops in C, including **for loops**, **while loops**, and **do-while loops**, are essential constructs for executing a block of code repeatedly until a specified condition is met.

1. **For Loops**: For loops are used when the number of iterations is known beforehand. They consist of an initialization, condition, and iteration expression, allowing precise control over the loop's execution.

2. **While Loops**: While loops execute a block of code as long as a specified condition is true. They are condition-controlled, meaning they continue until the condition evaluates to false.

   An iteration method that is best used when you don't know the number of iterations ahead of time.

3. **Do-While Loops**: Do-while loops are similar to while loops but guarantee that the block of code is executed at least once before checking the condition. This is useful when you want to ensure initialization or execution of certain code before checking the condition. **Input validation:** For scenarios where user input needs

validation, a `do-while` loop can ensure that the input validation process runs at least once. This prevents errors caused by invalid input and prompts the user to re-enter correct data

Types of Loops

Entry Control Loops: These loops check the condition before executing the statement. If the condition is false initially, the loop won't run.

- Examples: `for loop`, `while loop`.
- **For loop:** Executes a block of code for a fixed number of iterations.
- **While loop:** Repeats a block of code while a specified condition is true.

Exit Control Loops: These loops check conditions after executing statements. The loop runs at least once, even if the condition is initially false.

- Example: `do while loop`.
- **Do while loop:** Executes a block of code at least once, then repeats it while a specified condition is true.

Parts of a Loop:

A loop consists of four essential parts:

1. **Initialization:** Setting initial values or conditions before entering the loop.
2. **Conditions:** Evaluating conditions that determine loop execution.
3. **Statements:** Executing commands or statements inside the loop.
4. **Incrementation or Decrementation:** Adjusting loop control variables to progress through iterations.

While Loop

The `while loop` in C is straightforward and efficient, evaluating a condition before executing the loop body. It continues executing the loop body as long as the condition remains true. The loop continues until the condition becomes false, then execution moves to the statement following the loop.

- **Syntax:**

```
initialization;
while (condition) {
    statement;
    incrementation or decrementation;
}
```

- **Characteristics:**
    - An entry control loop.
    - The loop condition is evaluated before executing the loop body.
    - Executes statements as long as the condition remains true.

Example Consider a program to calculate and print the sequence from 1 to N:

```
int i, n;
printf("Enter the value of N: ");
scanf("%d", &n);
i = 1;
while (i <= n) {
    printf("%d, ", i);
    i++;
}
printf("\b "); // Removes the last comma
```

- Loop condition is a boolean expression, which evaluates to either true (1) or false (0).
- Be cautious not to place semicolons after the `while` keyword, which can lead to unexpected infinite loops.

Example

Consider a program to accept scores, calculate their sum and average, then print the results.

```
#include<stdio.h>
#include<conio.h>

int main() {
    int count = 0;
    float num = 0, sum = 0, avg = 0;

    printf("Enter score (-1 to stop): ");
    scanf("%f", &num);

    while (num >= 0) {
```

```
        sum += num;
        count++;
        printf("Enter score (-1 to stop): ");
        scanf("%f", &num);
    }

    avg = sum / count;
    printf("\nAverage = %f", avg);
    printf("\nSum = %f\n", sum);

    system("pause");
    return 0;
}
```

In this example, the loop continues until a negative value is entered, indicating the end of score input.

Do While Loop

The `do while` loop in C executes the loop body at least once before evaluating the loop condition. It's an exit control loop that ensures the loop body runs at least once.

- **Syntax:**

```
initialization;
do {
    statement;
    incrementation or decrementation;
} while (condition);
```

- **Characteristics:**
    - Executes the loop body before checking the condition.
    - Ensures the loop body runs at least once.

Example

Consider an example of a `do while` loop printing the sequence from 5 to 1:

```
int j = 5;
printf("Start\n");
do {
    printf("j = %i\n", j--);
```

```
} while (j > 0);
printf("Stop\n");
```

Output:

```
Start
j = 5
j = 4
j = 3
j = 2
j = 1
Stop
```

*Note

> Be cautious not to place semicolons after the `while` keyword, as it can lead to unexpected infinite loops.

For Loop in C

The `for` loop in C is a control structure used to iterate over a block of code multiple times. It provides a more concise way to control loop execution compared to `while` or `do-while` loops. Let's break down and refine the key aspects of the `for` loop:

1. **Structure of the For Loop:**
    - The `for` loop is constructed from a control statement that dictates how many times the loop will run and a command section containing one or more statements.
    - It has three parts: initialization, condition, and increment or decrement.
    - The syntax of the `for` loop is as follows:

```
for (initialize; condition; increment or decrement)
{
    statement;
}
```

2. **Initialization:**

- The initialization part is executed only once at the start of the loop.
- It initializes loop variables and sets up initial conditions.

3. **Condition:**

   - The condition is evaluated before each iteration of the loop.
   - If the condition is true, the loop continues; otherwise, it terminates.
   - This part determines whether the loop should continue running.

4. **Increment or Decrement:**

   - The increment or decrement part is executed after each iteration of the loop.
   - It modifies loop variables to control the loop's progress.

5. **Example:**

   - Here's an example demonstrating the use of a `for` loop to print numbers from 0 to 99:

```
#include<stdio.h>
#include<conio.h>
main()
{
    int i = 0;
    for(i = 0; i < 100; i++)
        printf("\ni=%d", i);
    system("pause");
}
```

   - This example initializes `i` to 0, runs the loop as long as `i` is less than 100, and increments `i` after each iteration.

6. **Notes:**

   - Standard headers like `stdio.h` and `conio.h` are commonly used in C programs for input/output operations and console I/O, respectively.
   - It's essential to remember the semicolons that separate the three parts of the `for` loop construct.
   - The `for` loop allows for more concise loop control, especially when the number of iterations is known in advance.

7. **Delay Loop:**

- A delay loop can be set up using a null statement. For example:

```
for (i = 100; i > 0; i--);
```

- This loop delays execution based on the value of `i`.

Understanding and mastering the `for` loop is crucial for efficient C programming as it offers a compact and powerful way to control repetitive tasks.

Understanding Loop Control and Jump Statements in C

In C programming, loops are fundamental for executing a set of statements repeatedly until a certain condition is met. Jump statements in C provide control flow alteration within loops.

There are four jump statements:

- **break:** Exits the loop immediately.
- **continue:** Skips the rest of the current iteration and proceeds to the next iteration.
- **goto:** Jumps from one part of the program to another, but usage is discouraged due to its potential to create complex logic.
- **return:** Used in functions to return a value and exit the function.

However, there are situations where you might need to alter the flow of execution within a loop or even break out of it entirely. This is where jump statements come into play. Let's explore them in detail:

1. **Break Statement**: Used within a loop, `break` immediately exits the loop, transferring control to the statement following the loop. It's commonly employed to terminate a loop prematurely.

    - Example:

```
while (scanf("%i", &j) != 1) {
    while ((ch = getchar()) != '\n');
    break; // Exit the loop
}
```

2. **Continue Statement**: When a certain condition is met, `continue` allows you to skip the remaining statements in the current iteration of the loop and proceed to the next iteration.
    - Example:

```
while ((ch = getchar()) != '\n') {
    if (ch == '.') continue; // Skip processing for '.'
    putchar(ch);
}
```

3. **Goto Statement**: Though available, using `goto` to jump from one part of the program to another is discouraged due to its potential to complicate logic and make code unreadable.

4. **Return Statement**: In functions, `return` not only sends a value back to the caller but also jumps from the current function definition to the calling function definition.

Remember, while these statements offer flexibility, they should be used judiciously to maintain code clarity and readability. Overuse, especially of `goto` and `continue`, can lead to spaghetti code.

Recap:

1. **Break Statement**: Used to exit a loop immediately. It's particularly useful for breaking out of infinite loops or prematurely terminating a loop when a certain condition is met. However, excessive use can lead to code complexity.
2. **Continue Statement**: Skips the rest of the current iteration of the loop and proceeds to the next iteration. Unlike `break`, it doesn't terminate the loop but rather jumps to the loop's condition for evaluation. Overusing `continue` can also make code harder to understand.
3. **Goto Statement**: Allows jumping from one part of the program to another, including exiting from deeply nested loops. However, it's generally discouraged due to its potential to create spaghetti code and make logic convoluted.
4. **Return Statement**: Primarily used in functions to exit the function and optionally return a value to the calling code. It's not a loop control statement but is included here for completeness.
5. **Exit Function**: Terminates the entire program execution immediately. It's different from other jump statements as it doesn't control the flow within loops but rather exits the entire program.

Understanding when and how to use these jump statements is crucial for writing clear, efficient, and maintainable code. Introduction to C Programming Looping Constructs

Lab-0004-Control-Statements-Looping

C Programming Journey