C++ Programming Fundamentals: Week 4 - Control Flow

Welcome to Week 4 of our C++ programming journey! This week, we're diving into the essential world of control flow. Understanding how to control the execution path of your programs is fundamental to building dynamic, intelligent, and responsive applications. We'll explore comparison operators, logical operators, and various conditional statements that empower your code to make decisions.

Think of control flow as the traffic cop of your program. It directs where the program goes next based on certain conditions. Mastering these concepts will allow your C++ programs to adapt to different situations, process user input intelligently, and solve more complex problems. Let's get started on giving your code the power to decide!

Comparison Operators: The Decision Makers

Comparison operators are the backbone of decision-making in programming. They allow you to compare two values and determine their relationship. The result of a comparison operation is always a boolean value: either `true` (meaning the condition is met) or `false` (meaning it's not).

These operators are crucial for checking conditions like whether a user entered the correct password, if a number is within a certain range, or if two variables hold the same value. They form the basis for all conditional logic you'll write.

Equal To (==)

Checks if two values are equal. For example, `x == 5` would be true if x is 5.

Not Equal To (!=)

Checks if two values are not equal. For example, `x != 10` would be true if x is not 10.

Greater Than (>)

Checks if the left value is greater than the right. For example, `age > 18`.

Less Than (<)

Checks if the left value is less than the right. For example, `score < 100`.

Greater Than or Equal To (>=)

Checks if the left value is greater than or equal to the right. For example, `grade >= 60`.

Less Than or Equal To (<=)

Checks if the left value is less than or equal to the right. For example, `temp <= 32`.

Comparison Operators: Code Example

Let's see comparison operators in action. This simple C++ program demonstrates how each operator evaluates different conditions.

```
#include <iostream>

int main() {
    int num1 = 10;
    int num2 = 5;

    std::cout << "Comparing " << num1 << " and " << num2 << std::endl;

    // Equal To (==)
    std::cout << "num1 == num2: " << (num1 == num2) << std::endl; // Output: 0 (false)

    // Not Equal To (!=)
    std::cout << "num1 != num2: " << (num1 != num2) << std<< "::endl; // Output: 1 (true)

    // Greater Than (>)
    std::cout << "num1 > num2: " << (num1 > num2) << std::endl; // Output: 1 (true)

    // Less Than (<)
    std::cout << "num1 < num2: " << (num1 < num2) << std::endl; // Output: 0 (false)

    // Greater Than or Equal To (>=)
    std::cout << "num1 >= num2: " << (num1 >= num2) << std::endl; // Output: 1 (true)

    // Less Than or Equal To (<=)
    std::cout << "num1 <= num2: " << (num1 <= num2) << std::endl; // Output: 0 (false)

    return 0;
}
```

In C++, a boolean `true` is often represented by `1` and `false` by `0` when printed to the console. Notice how the output clearly shows the result of each comparison.

Logical Operators: Combining Conditions

Logical operators allow you to combine multiple comparison expressions, creating more complex conditions. They are essential for situations where a decision depends on several factors being true or false simultaneously.

Imagine you need to check if a user is both logged in AND has administrative privileges. Or perhaps if a discount applies if the customer is a student OR a senior citizen. Logical operators provide the syntax for these nuanced scenarios.

Logical AND (&&)

Returns `true` if BOTH conditions are `true`. If even one condition is `false`, the entire expression is `false`.

```
condition1 && condition2
```

Logical OR (||)

Returns `true` if AT LEAST ONE condition is `true`. The entire expression is only `false` if both conditions are `false`.

```
condition1 || condition2
```

Logical NOT (!)

Reverses the boolean value of a condition. If a condition is `true`, `!` makes it `false`, and vice versa.

```
!condition
```

Logical Operators: Code Example

Here's a C++ example showcasing how logical operators evaluate multiple conditions.

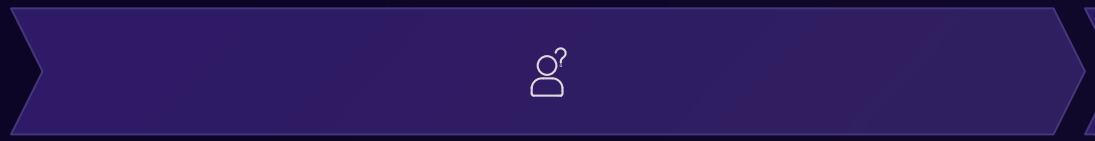
```
#include <iostream>

int main() {
    int age = 25;
    bool isStudent = true;
    bool hasLicense = false;

    // Logical AND (&&)
    // Check if eligible for a student discount AND is an adult
    if (isStudent && age >= 18) {
        std::cout << "Eligible for student discount!" << std::endl; // This will execute
    }

    // Logical OR (||)
    // Check if eligible to drive (either has license OR is old enough for a permit)
    if (hasLicense || age >= 16) {
        std::cout << "Can learn to drive or already drives." << std::endl; // This will execute
    }

    // Logical NOT (!)
    // Check if not a student
    if (!isStudent) {
        std::cout << "Not a student." << std::endl; // This will NOT execute
    }

    // Combining multiple conditions
    if ((age > 20 && age < 30) && !hasLicense) {
        std::cout << "You are in your 20s and don't have a license." << std::endl; // This will execute
    }

    return 0;
}
```

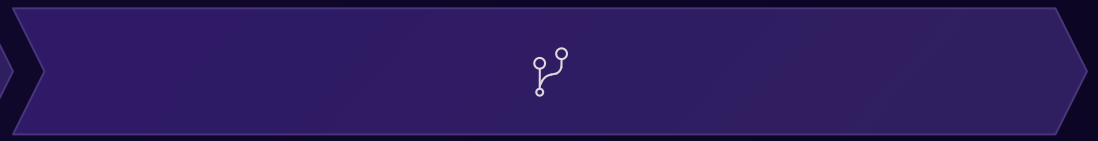
Using parentheses `()` helps clarify the order of operations, especially when combining different logical operators. It's like grouping mathematical expressions.

Conditional Statements: If, If-Else

Conditional statements are programming constructs that execute different blocks of code based on whether a specified condition evaluates to `true` or `false`. They are the fundamental tools for making your programs truly interactive and dynamic.

The `if` statement is the simplest form, executing code only if its condition is met. The `if-else` statement provides an alternative path, ensuring that one of two blocks of code will always execute.





The `if` Statement

Executes a block of code only if the specified condition is `true`. If the condition is `false`, the code block is skipped entirely.

```
if (condition) { // code to execute }
```

The `if-else` Statement

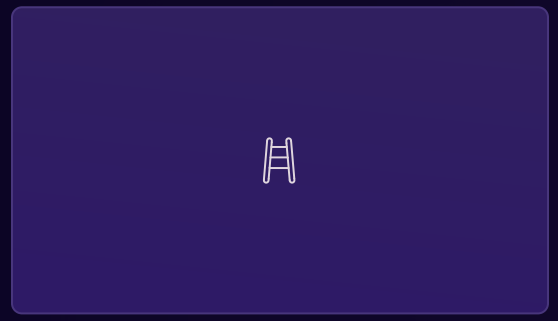
Provides two execution paths. If the condition is `true`, the `if` block runs. Otherwise (if `false`), the `else` block runs.

```
if (condition) { // true code } else { // false code }
```

Conditional Statements: If-Else-If, Nested If

When you have more than two possible outcomes, the `if-else-if` ladder comes to your rescue. For even more intricate decision-making, `nested if` statements allow you to check conditions within other conditions.

These structures enable your programs to handle a wide range of scenarios, from grading systems to complex game logic.



The `if-else-if` Ladder

Allows you to check multiple conditions sequentially. The first condition that evaluates to `true` will have its corresponding code block executed, and the rest are skipped.

`if (cond1) { ... } else if (cond2) { ... } else { ... }`

Nested `if` Statements

An `if` statement (or an `if-else` statement) placed inside another `if` or `else` block. Useful for conditions that depend on a prior condition being met.

`if (outer_cond) { if (inner_cond) { ... } }`

Conditional Statements: Code Example

Let's combine all these conditional statements into a single, illustrative program:

```
#include <iostream>

int main() {
 int score = 75;
 bool isMember = true;

 // Simple if statement
 if (score >= 60) {
 std::cout << "You passed the exam." << std::endl;
 }

 // if-else statement
 if (score >= 90) {
 std::cout << "Excellent work!" << std::endl;
 } else {
 std::cout << "Keep practicing to achieve excellence." << std::endl;
 }

 // if-else-if ladder
 if (score >= 90) {
 std::cout << "Grade: A" << std::endl;
 } else if (score >= 80) {
 std::cout << "Grade: B" << std::endl;
 } else if (score >= 70) {
 std::cout << "Grade: C" << std::endl;
 } else if (score >= 60) {
 std::cout << "Grade: D" << std::endl; // This will execute for score = 75
 } else {
 std::cout << "Grade: F" << std::endl;
 }

 // Nested if statement
 if (isMember) {
 if (score >= 80) {
 std::cout << "As a member with a high score, you get a special badge!" << std::endl;
 } else {
 std::cout << "As a member, you have access to extra study materials." << std::endl; // This will execute
 }
 } else {
 std::cout << "Consider becoming a member for more benefits." << std::endl;
 }

 return 0;
}
```

Summary: Building Intelligent Programs

This week, we've equipped ourselves with the fundamental tools for creating programs that can make decisions and react to different inputs and conditions. From the basic comparisons to complex logical combinations and multi-branched conditional statements, you now have the power to direct your program's flow.

Mastering these concepts is a critical step towards writing more sophisticated, user-friendly, and effective C++ applications. Keep practicing, and soon you'll be building programs that think for themselves!

1

Comparison Operators

Used to compare values and return `true` or `false`.

2

Logical Operators

Combine multiple conditions (`&&`, `||`, `!`).

3

Conditional Statements

Execute code blocks based on conditions (`if`, `if-else`, `if-else-if`, `nested if`).

Practice Questions

Time to test your understanding! Work through these questions to solidify your knowledge of comparison operators, logical operators, and conditional statements.

1. Write a C++ program that asks the user to enter two integers. Use an `if-else` statement to determine and print which number is larger, or if they are equal.
2. Create a program that prompts the user for their age and whether they have a driver's license (input 'y' for yes, 'n' for no). Use logical operators and conditional statements to print:
    - "You can drive!" if they are 18 or older AND have a license.
    - "You can get a permit soon!" if they are between 16 and 17 (inclusive) AND do NOT have a license.
    - "You are too young to drive." otherwise.
3. Develop a C++ program that simulates a simple grading system. Ask the user for a percentage score (0-100). Use an `if-else-if` ladder to assign and print a letter grade (A, B, C, D, F) based on standard grading scales (e.g., 90-100 A, 80-89 B, etc.).
4. Explain in your own words the difference between the `=` (assignment operator) and `==` (comparison operator). Provide a small code snippet demonstrating each.