Chapter 3: Control Flow and Logic

Control flow statements are the decision-making backbone of a program. They allow you to steer the path of execution based on conditions and to repeat actions, turning a simple top-to-bottom script into a dynamic and responsive algorithm.

3.1 Selection Statements (If/Switch)

Why do we need selection? Selection statements allow a program to respond to different inputs or states. `if` is the general-purpose tool for handling varied boolean conditions, while `switch` is a specialized tool for efficiently comparing a single value against multiple constant possibilities.

`if-else` Statements

This is the most fundamental decision-making construct. It executes a block of code if a condition is true, and can execute alternative blocks for other conditions (`else if`) or a default case (`else`).

C++17: `if` with Initializer

Why was this added? It solves a common code smell. Often, you need a variable only for the duration of an `if-else` check (like an iterator or a return code). Before C++17, you had to declare this variable in the surrounding scope, where it could be seen and modified even after the `if` statement was finished. This new syntax elegantly tightens the variable's scope to just the `if-else` block, which is a core principle of good C++ design: keep variables in the narrowest scope possible.

```cpp
// Old way: `it` leaks into the outer scope
{
    auto it = my_map.find("key");
    if (it != my_map.end()) {
        // use it...
    }
} // `it` is gone now, but we needed an extra set of braces.

// C++17 way: `it` only exists inside the if and else blocks. Cleaner and safer.
if (auto it = my_map.find("key"); it != my_map.end()) {
    // use it...
} else {
    // `it` is also in scope here!
}
// `it` is out of scope here.
```

`switch` Statements

What are the rules? 1. The condition variable in a `switch` must be of an integral type (e.g., `int`, `char`, `enum`). 2. `case` labels must be compile-time constants. 3. Execution will **fall through** from one case to the next unless you explicitly use a `break` statement.

Fallthrough: While often a source of bugs, intentional fallthrough can be useful for grouping cases. C++17 introduced the `[[fallthrough]]` attribute to make this intent clear to the compiler and other programmers.

```cpp
char c = 'a';
switch (c) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        std::cout << "Vowel" << std::endl;
        break;
    case 'z':
        std::cout << "This is the last letter." << std::endl;
        [[fallthrough]]; // C++17: I know I'm falling through!
    default:
        std::cout << "Consonant or other character." << std::endl;
        break;
}
```

Optimization Insight: Jump Tables For a dense range of integer cases (e.g., case 1, 2, 3, 4, 5…), compilers can perform a powerful optimization. Instead of a series of comparisons (`if (x==1)... if (x==2)...`), it builds a **jump table**—an array of memory addresses. The value of the switch variable is used as an index into this array, allowing the CPU to jump *directly* to the correct code block in a single operation, which is much faster than a long `if-else` chain.

3.2 Loop Statements and Range-Based For

Why use loops? Loops automate repetitive tasks. The different types of loops give you flexibility in defining the start and end conditions for that repetition.

- **`while` loop:** The most basic loop. The condition is checked *before* each iteration. Use it when you don't know how many iterations you'll need, but you know the condition to stop.
- **`do-while` loop:** The condition is checked *after* each iteration. This guarantees the loop body will execute at least once. Useful for things like prompting for user input.
- **`for` loop:** The classic C-style loop. It consolidates the initialization, condition, and increment expressions in one line. Use it when you have a clear start, end, and increment step (e.g., iterating through an array by index).
- **Range-Based `for` loop (C++11):** The modern, preferred way to iterate over an entire collection. It is safer and more readable than a traditional `for` loop as it eliminates the possibility of off-by-one errors.

How does the range-based `for` loop work? It is syntactic sugar for a loop that uses iterators. The compiler transforms it into something like this:

```
std::vector<int> numbers = {1, 2, 3};

// You write this:
for (int num : numbers) { /* ... */ }

// The compiler generates something like this:
{
    auto&& __range = numbers;
    auto __begin = __range.begin();
    auto __end = __range.end();
    for (; __begin != __end; ++__begin) {
        int num = *__begin;
        /* ... */
    }
}
```

This shows that it works with anything that provides `begin()` and `end()` methods that return iterators.

Critical Concept: Short-Circuit Evaluation

Why is this important? This is not just an optimization; it is a guaranteed behavior that is a cornerstone of C++ safety idioms.

When evaluating `A && B` or `A || B`, the language guarantees that the left side `A` is evaluated first. Then, a decision is made: * For `A && B`: If `A` is `false`, the result must be `false`, so **`B` is never evaluated**. * For `A || B`: If `A` is `true`, the result must be `true`, so **`B` is never evaluated**.

How is this used? This is critical for preventing errors, most commonly for checking a pointer before dereferencing it.

```
// This pattern is fundamental to safe C++
// If `ptr` is null, the expression short-circuits and ptr->isReady() is never called,
// preventing a crash.
bool check_status(Widget* ptr) {
    return (ptr != nullptr) && (ptr->isReady());
}
```

Projects for Chapter 3

Project 1: The Number Guessing Game

- **Problem Statement:** Write a program that generates a random number between 1 and 100. Use a `do-while` loop to repeatedly prompt the user to enter a guess. Inside the loop, use `if-else if-else` statements to tell the user if their guess is too high, too low, or correct. The loop should terminate only when the user guesses the number correctly. Keep track of the number of guesses.
- **Core Concepts to Apply:** Loops (`do-while` is a good fit), selection (`if-else`), handling user input (`std::cin`), and random number generation (`<random>`).
- **Hint:** The `do-while` loop ensures the user is prompted for input at least once.

Project 2: The `switch`-based Text Adventure

- **Problem Statement:** Create a very simple text-based adventure game. The player is in a room and can choose to go `'n'` (north), `'s'` (south), `'e'` (east), or `'w'` (west). Use a `while` loop to keep the game running and a `switch` statement inside to handle the user's character input. Each case should print a different message (e.g., "You walk north and find a treasure chest!", "You walk south and hit a wall."). Have a case for quitting (`'q'`).
- **Core Concepts to Apply:** `switch` statement for character-based choices, `while` loop for the main game loop, `char` input.
- **Hint:** Remember to use `break` in your `case` blocks! Use a `default` case to handle invalid user input.

Project 3: Vector Statistics Calculator

- **Problem Statement:** Create a `std::vector<double>` and fill it with some numbers. Using a single range-based `for` loop, calculate the sum, mean (average), minimum, and maximum values in the vector. Print the results. First, write the loop using `for (double value : my_vector)`. Then, change it to `for (const double& value : my_vector)`. Add comments to your code explaining why the second version is often preferred, especially when the vector contains large objects instead of simple `doubles`.
- **Core Concepts to Apply:** Range-based `for` loop, `std::vector`, iteration by value vs. iteration by `const` reference.
- **Hint:** You will need to declare your `sum`, `min`, and `max` variables before the loop starts. Initialize `min` to a very large number and `max` to a very small number.