C++ Selection Statements

CSI-140 Introduction to Programming

Instructor: Dr. Vikas Thammanna Gowda Semester: Fall 2025

Introduction

In programming, **selection/control statements** allow the flow of execution to change based on certain conditions or choices. This means the program can make decisions and execute certain parts of code only when specific conditions are met. In C++, two primary decision-making (selection) control structures are the `if-else` statement and the `switch` statement. Using these, a program can choose different paths of execution. For example, a program can check a user's input and decide what to do next (such as granting access if a password is correct, or showing an error if not).

In the following notes, we will explore how to use `if-else` (including simple `if`, `if-else` pairs, multiple `else if` conditions, and nested `if` statements) and the `switch` statement in C++. We will also discuss common pitfalls (mistakes) to avoid when using these control statements. Each concept will be explained with simple examples that a high school student can easily follow.

The `if-else` Statement

An `if` statement allows a program to execute a certain section of code only if a specified condition is true. If the condition is false, the code inside the `if` block is skipped. An optional `else` part can be used to execute alternate code when the condition is false. We will look at different forms of `if-else` usage:

Simple `if` Statement

A simple `if` statement has the form:

Syntax:

```
if (condition) {
    statements
}
```

When the condition is true (non-zero), the statements inside the braces { } are executed. If the condition is false (zero), those statements are skipped and the program continues after the `if` block. There is no `else` part in a simple `if`, so nothing happens when the condition is false.

The condition inside the `if` is usually a comparison or logical test that evaluates to `true` or `false` (or in C++ an integer expression where 0 is false and non-zero is true). For example, `x > 0`, `x == 10`, or `y <= 5` are typical conditions.

Example: Simple `if` In the following example, the program asks the user for their age and uses an `if` statement to check if the age is 18 or above. If the condition is true (the user is at least 18 years old), it will print a message. If the condition is false (user is younger than 18), the `if` block is skipped (so no eligibility message is printed). In either case, the program then prints a thank-you message:

Example: Simple if statement

```
#include <iostream>
using namespace std;

int main() {
    int user_age;
    cout << "Enter your age: ";
    cin >> user_age;
    if (user_age >= 18) {
        cout << "You are eligible to vote." << endl;
    }
    cout << "Thank you for using the program." << endl;
    return 0;
}
```

Let's break down this example:

- The condition in the `if` is `user_age >= 18`. If the user enters 18 or a higher number, the condition is true and the program executes the code inside the `if` block, printing `"You are eligible to vote."`.
- If the user enters, say, 16, the condition `user_age >= 18` is false. In this case, the program skips the `if` block entirely and directly goes to the next statement after it. So it would only print `"Thank you for using the program."` and nothing about voting eligibility.
- The output for an input of 20 would be:

Output:

```
Enter your age: 20
You are eligible to vote.
Thank you for using the program.
```

For an input of 16, the output would be:

Output:

```
Enter your age: 16
Thank you for using the program.
```

Common Pitfalls for `if`:

- **Using `=` instead of `==`:** Remember that `==` is the comparison operator (to check equality) while `=` is the assignment operator. Writing `if (x = 5)` will assign 5 to `x` (which is a bug) instead of comparing `x` to 5. The assignment `x = 5` will actually set `x` to 5 and then the condition will be true if 5 is non-zero. Always use `==` for comparison.
- **Forgetting braces for multiple statements:** If you have more than one statement to execute when the condition is true, you must enclose them in { } braces. If you forget the braces, C++ will only treat the next single statement as part of the `if`. This can lead to logic errors. It is a good practice to use braces even for a single statement to avoid mistakes if you add more code later.

- **Semicolon right after the condition:** Do not put a semicolon immediately after the `if` condition. For example, `if (x > 0); { cout << "Positive"; }` is incorrect. The semicolon ends the `if` statement prematurely, so the block in braces becomes unrelated and will execute regardless of the condition. This bug can be hard to spot.

`if-else` (Two-Way Selection)

A simple `if` covers the case when a condition is true. If we want to execute an alternative block of code when the condition is false, we use an `else` clause. The syntax is:

Syntax:

```
if (condition) {
    yes (true) statements
} else {
    no (false) statements
}
```

With this structure, one of two blocks will execute: if the condition is true, the first block runs; if the condition is false, the block after the `else` runs. This is a two-way decision.

Example: `if-else` The following snippet checks whether a given number is even or odd using an `if-else` statement. Depending on the result of the condition, it prints one of two messages:

Example: if-else statement

```
#include <iostream>
using namespace std;

int main() {
    int input_number;
    cout << "Enter an integer: ";
    cin >> input_number;
    if (input_number % 2 == 0) {
        cout << input_number << " is even." << endl;
    } else {
        cout << input_number << " is odd." << endl;
    }
    return 0;
}
```

Here:

- The condition `input_number % 2 == 0` checks if `input_number` is divisible by 2 with no remainder (which means `input_number` is even).
- If the condition is true (for example, `input_number` is 8), the first `cout` in the `if` block executes, printing "8 is even.".
- If the condition is false (`input_number` is not divisible by 2, e.g., `input_number = 7`), the `else` block executes and prints "7 is odd.".
- Exactly one of the two blocks will run, and then the program continues after the `if-else`.

Using `if-else` ensures that one of the outcomes happens (either even-case or odd-case in this example), which is often what we want for mutually exclusive cases.

Common Pitfalls for `if-else`:

- **Misplacing the `else`:** An `else` is always associated with the nearest previous `if` that doesn't already have an `else`. Using braces consistently helps avoid confusion about which `if` an `else` belongs to.
- **Omitting braces in complex statements:** If you write `if` and `else` without braces, only one statement after each will be considered in that branch. For example:

Example: Misleading indentation without braces

```
if (x > 0)
    cout << "Positive";
    cout << "Number";
else
    cout << "Negative";
```

This code is misleadingly indented. In reality, if `x > 0` only "Positive" is printed, but "Number" will always print because it's not in the `if`. This is likely a mistake in intent. Always using braces can prevent this kind of error.

Multiple Conditions: `if-else-if` Ladder

Sometimes, you need to check several conditions one after another and choose one action out of many possibilities. This can be done by chaining multiple `if` statements with `else if` clauses. This is commonly called an "`if-else-if ladder`" or a "chain of if-else statements."

The structure is:

Syntax:

```
if (condition1) {
    // code if condition1 is true
} else if (condition2) {
    // code if condition1 is false and condition2 is true
} else if (condition3) {
    // code if condition1 and condition2 are false and condition3 is
       true
} else {
    // code if none of the above conditions are true
}
```

The program will evaluate each condition in order:

- If condition1 is true, it executes that block and skips the rest of the ladder.
- If condition1 is false, it then checks condition2. If condition2 is true, it executes the second block and skips the rest.
- This continues down the chain. If none of the conditions is true, the final `else` block (if provided) will execute.

Only one of the blocks in an `if-else-if` chain will run at most (or none, if there is no final `else` and all conditions are false).

Example: Grading System (Multiple `else if`) Consider a grading system where a score from 0 to 100 needs to be converted to a letter grade:

- 90 and above is grade `A`
- 80–89 is grade `B`
- 70–79 is grade `C`
- 60–69 is grade `D`
- below 60 is grade `F`

We can write an `if-else-if` ladder to determine the grade based on the score:

Example: if-else-if ladder for grading

```
#include <iostream>
using namespace std;

int main() {
    int student_score;
    cout << "Enter your score (0-100): ";
    cin >> student_score;
    char letter_grade;
    if (student_score >= 90) {
        letter_grade = 'A';
    } else if (student_score >= 80) {
        letter_grade = 'B';
    } else if (student_score >= 70) {
        letter_grade = 'C';
    } else if (student_score >= 60) {
        letter_grade = 'D';
    } else {
        letter_grade = 'F';
    }
    cout << "Your grade is " << letter_grade << "." << endl;
    return 0;
}
```

Explanation:

- If `student_score >= 90`, the first condition is true and `letter_grade` is set to 'A'. The rest of the `else if` statements are skipped.
- If `student_score` is 75, the first condition is false, the program then checks `student_score >= 80` (false), then checks `student_score >= 70` which is true, so it sets `letter_grade = 'C'`. It will then skip the remaining conditions and the final `else`.
- If `student_score` is 45, all the conditions `>= 90`, `>= 80`, `>= 70`, `>= 60` will be false, so the program will go to the `else` block and assign 'F'.

Common Pitfalls for multiple `if-else-if`:

- **Order of conditions matters:** Once a condition is true in the chain, the rest are not checked. So you should order from most specific to more general conditions or vice versa appropriately. For example, if we had checked `student_score >= 60` in the first `if`, that would also be true for scores like 85, and those would never reach the later check for 80 or 90. Thus, the order should be from highest to lowest in this grading example.
- **Missing a final `else`:** If you don't provide a final `else` to catch "none of the above" cases, some input values might not trigger any action. This could be fine in some scenarios, but often you want to handle all possible cases (for example, by providing an `else` that prints an error for an invalid score or a score out of range).

Nested `if` Statements

"**Nested**" `if` statements mean an `if` statement inside another `if` (or inside an `else` block). Nesting allows us to check a second condition only when the first condition is true (or false, depending on where you nest it). This is useful for making more specific decisions after a general decision.

Every `if` must be paired with its own `else` (if an else is needed) inside the block it belongs to. By using braces and proper indentation, nested `if` statements can be made clear.

Example: Nested `if` Let's say we want to categorize a number as positive, negative, or zero, and additionally say whether a positive number is even or odd. We can use nested `if` statements to do this:

Example: Nested if statements

```
#include <iostream>
using namespace std;

int main() {
    int input_value;
    cout << "Enter an integer: ";
    cin >> input_value;
    if (input_value >= 0) {      // First level: non-negative check
        if (input_value == 0) {   // Second level: check for zero
            cout << "The number is zero." << endl;
        } else {                  // input_value is positive (greater
           than 0)
            cout << "The number is positive." << endl;
            if (input_value % 2 == 0) {    // Third level: check even
               /odd for positive
                cout << "It is even." << endl;
            } else {
                cout << "It is odd." << endl;
            }
        }
    } else {
        cout << "The number is negative." << endl;
    }
    return 0;
}
```

In this nested structure:

- The outermost `if` checks if `input_value >= 0`. If this is true, we enter the outer if-block; if false, we skip to the `else` and report the number is negative. (Notice the `else` at the bottom corresponds to the first `if`.)
- Inside the outer `if` block, we have another `if`: `if (input_value == 0)`. This checks for the special case of zero. If `input_value` is exactly 0, it prints "number is zero." and skips the `else` that follows.
- The `else` that pairs with `if (input_value == 0)` handles the case where `input_value` is not zero. Since we are inside the outer if, we already know `input_value` is non-negative, and not zero, so it must be positive. Thus, it prints "number is positive.".
- Inside that else (which runs for any positive number), we nest a third `if` to check if the positive `input_value` is even or odd using `input_value % 2 == 0`. Depending on the result, it prints "even" or "odd".

For example, if `input_value = 0`, the output will be:

Output:

```
The number is zero.
```

If `input_value = 7`, the output will be:

Output:

```
The number is positive.
It is odd.
```

If `input_value = -4`, the output will be:

Output:

```
The number is negative.
```

Common Pitfalls for nested `if`:

- **Dangling `else`:** Always use braces to avoid confusion about which `if` an `else` pairs with in nested situations. C++ will by default pair an `else` with the closest previous unmatched `if`. If braces are omitted in complex nesting, you might think an `else` corresponds to an outer `if` when it actually paired with an inner one.
- **Deep nesting and readability:** Too many nested levels can make code hard to read and debug. If you find yourself nesting a lot, consider whether you can simplify the logic (for example, by using logical operators like `&&` or `||` in a single `if` condition, or by restructuring the conditions).

`switch` Statement

The `switch` statement is another way to control flow based on the value of a variable or expression. It is especially useful when you have a single variable that can equal several different constant values and you want to execute different code for each value. A `switch` can often make multi-way branching logic clearer than a long chain of `if-else-if` for this particular scenario.

The general form of a `switch` in C++ is:

Syntax:

```
switch (expression) {
    case value1:
        // code to execute if expression == value1
        break;
    case value2:
        // code to execute if expression == value2
        break;
    // (more case labels as needed)
    default:
        // code to execute if none of the above cases match
}
```

How it works:

- The program evaluates the `expression` in the `switch` once.
- It then compares the result to each `case` label (value1, value2, etc.) in order.
- If a matching value is found, the program executes the code starting from that `case`. It continues executing until a `break` statement is encountered.
- The `break` statement causes the program to jump out of the `switch` structure, skipping the remaining cases.
- If no case value matches the expression, the `default` section (if provided) will execute. The `default` is like an "else" for the switch, handling any values that weren't matched by a case.

One key rule: the values for case labels must be constant expressions (like literal numbers or characters, or `const` values known at compile time), and the switch expression itself must be an integer type (such as `int`, `char`, or an `enum`). For example, you cannot switch on a `std::string` or a `float` directly in standard C++.

Example: Days of the Week using `switch` The following program takes a number (1 through 7) representing a day of the week and prints out the corresponding day name. This is a scenario with multiple discrete values where using a `switch` is convenient:

Example: switch statement for days of week

```
#include <iostream>
using namespace std;

int main() {
    int day_number;
    cout << "Enter a day number (1-7): ";
    cin >> day_number;
    switch (day_number) {
        case 1:
            cout << "Monday" << endl;
            break;
        case 2:
            cout << "Tuesday" << endl;
            break;
        case 3:
            cout << "Wednesday" << endl;
            break;
        case 4:
            cout << "Thursday" << endl;
            break;
        case 5:
            cout << "Friday" << endl;
            break;
        case 6:
            cout << "Saturday" << endl;
            break;
        case 7:
            cout << "Sunday" << endl;
            break;
        default:
            cout << "Invalid day number!" << endl;
    }
    return 0;
}
```

If the user inputs `3`, the program will match `case 3` and print "Wednesday". If the user inputs `7`, it will print "Sunday". If the input is `10` (which doesn't match any case 1-7), the program will jump to `default` and print "Invalid day number!".

Let's analyze important parts of this example:

- Each `case` label (1, 2, 3, ..., 7) represents a possible value for `day_number`. The code under that case executes if `day_number` matches the label.
- The `break` at the end of each case block tells the program to exit the switch after executing that case. If we omitted a `break`, the program would continue into the next case's code (this is called "fall-through"). For example, if `day_number = 3` and we had no `break` after printing "Wednesday", the program would then proceed to execute the code for case 4 ("Thursday") and onward until a break or the end of the switch is reached. In this example, such fall-through is not desired, so we include `break` for each case.
- The `default` case handles any value of `day_number` that isn't 1 through 7. It's like a safety

net for unexpected inputs.

Common Pitfalls for `switch`:

- **Forgetting `break`:** If you forget to put a `break` at the end of a case, the execution will fall through to the next case. Sometimes fall-through is intentional (advanced usage), but usually for beginners it causes logical errors. Always check that each case ends with `break` (or another flow control like `return`) unless you specifically want to execute combined cases.
- **Invalid case types:** Ensure that the values you use in `case` labels are of a compatible type with the switch expression and are constant. For example, using string literals in cases (like `case "yes":`) is not allowed. Also, switching on a floating-point number or a string is not allowed in standard C++.
- **Duplicate case values:** Each case value must be unique. If you accidentally duplicate a case label, the program will not compile.
- **Not using braces for multiple statements in cases:** In C++, you do not need braces for multiple statements under a case label (the switch handles grouping until a break). However, if you declare new variables in a case without braces, you may need to be careful with scope and initialization. A common error is writing code like:

```
switch(option) {
    case 1:
        int x;        // error: initialization skips case
        x = 5;
        break;
    case 2:
        ...
}
```


The above will cause a compilation error because of the way variable initialization is scoped across cases. The fix is to use braces to create a new scope for that case:

```
    case 1: {
        int x = 5;
        cout << x;
        break;
    }
```

But for simple cases where you just perform actions (as in our day example), this usually isn't an issue.

*

Conclusion In summary, `if-else` statements and `switch` statements are fundamental tools for making decisions in a C++ program. The `if-else` construct is very flexible and can handle a wide range of conditions, including complex logical expressions and nested decisions. The `switch` statement is a convenient way to branch on a single variable or expression that can take on a limited set of values.

When using these control statements, always be mindful of the syntax and the common pitfalls:

- Use `if-else` when checking ranges or multiple different conditions that don't all depend on one single variable.
- Use `switch` when you are comparing one variable to many constant values – it can make the code cleaner and more readable.
- Watch out for the common mistakes such as using `=` instead of `==`, forgetting braces or break statements, and improper ordering of conditions.

By understanding and practicing with these examples, students should become comfortable with controlling the flow of execution in their programs, allowing for more dynamic and correct program behavior based on different inputs and conditions.