Gorgo Starting C++

April 11, 2026

Contents

5. Control Flow

So far, every program you have written runs straight from top to bottom. Every line executes exactly once, in order. That is fine for simple tasks, but real programs need to make decisions and repeat actions. Control flow statements let your program choose which code to run and how many times to run it.

In this chapter, you will learn `if` statements for making decisions, loops for repeating code, and `switch` statements for selecting among multiple options. These are the building blocks that let your programs do interesting things.

`if` Statements

An `if` statement tests a condition and runs a block of code only when the condition is true:

```
#include <iostream>

int main() {
    int score = 85;

    if (score >= 90) {
        std::cout << "Excelente!\n";
    }

    return 0;
}
```

The condition inside the parentheses must be a boolean expression. If it evaluates to `true`, the code inside the braces runs. If it evaluates to `false`, the code is skipped entirely.

`else`

You can add an `else` block that runs when the condition is false:

```
int temperature = 30;

if (temperature > 100) {
    std::cout << "Too hot\n";
} else {
    std::cout << "Comfortable\n";
}
```

`else if`

When you have more than two possibilities, chain `else if` blocks together:

```
int score = 75;

if (score >= 90) {
    std::cout << "A\n";
} else if (score >= 80) {
    std::cout << "B\n";
} else if (score >= 70) {
    std::cout << "C\n";
} else {
    std::cout << "Try again\n";
}
```

The conditions are tested in order from top to bottom. As soon as one condition is true, its block runs and the rest are skipped. The final `else` catches anything that did not match an earlier condition.

Nested `if` Statements

You can put `if` statements inside other `if` statements:

```
int age = 20;
bool has_ticket = true;

if (age >= 18) {
    if (has_ticket) {
        std::cout << "Welcome to the show\n";
    } else {
        std::cout << "You need a ticket\n";
    }
} else {
    std::cout << "Must be 18 or older\n";
}
```

Guard Clauses

As you add more checks, nested `if` statements start drifting to the right side of the page and the real "success" path gets buried inside multiple pairs of braces. Imagine a function that has to check several things before letting someone into a show:

```
bool try_enter_show(int age, bool has_ticket, int seats_left) {
    if (age >= 18) {
        if (has_ticket) {
            if (seats_left > 0) {
                std::cout << "Welcome to the show\n";
                return true;
            } else {
                std::cout << "Sold out\n";
                return false;
            }
        } else {
            std::cout << "You need a ticket\n";
            return false;
        }
    } else {
        std::cout << "Must be 18 or older\n";
        return false;
    }
}
```

The success case ends up four levels deep, and each failure message sits far away from the condition that rejected it. Tracing why a particular message prints means counting open braces until you find the matching `else`.

A **guard clause** is an early `return` that rejects an invalid case as soon as you detect it, instead of wrapping the rest of the function in a `then` branch. Rewriting the same function with guard clauses:

```
bool try_enter_show(int age, bool has_ticket, int seats_left) {
    if (age < 18) {
        std::cout << "Must be 18 or older\n";
        return false;
```

```
    }
    if (!has_ticket) {
        std::cout << "You need a ticket\n";
        return false;
    }
    if (seats_left <= 0) {
        std::cout << "Sold out\n";
        return false;
    }
    std::cout << "Welcome to the show\n";
    return true;
}
```

Every failure case stands on its own at the same indentation level, the condition that triggers it sits right next to the message, and the happy path — the code that runs when *everything* checks out — lives at the bottom of the function, unindented. Readers no longer have to mentally stack up open braces to figure out which combination of conditions led to "Welcome to the show".

Tip: Deeply nested `if` statements make code hard to read. If you find yourself nesting more than two or three levels deep, consider using `else if` chains or guard clauses to flatten the logic.

Trap: A common mistake is using `=` (assignment) instead of `==` (comparison) in a condition:

```
int x = 0;
if (x = 5) {           // BUG: assigns 5 to x, then tests 5 (true)
    std::cout << "oops\n";  // always prints
}
```

The compiler may warn you about this. Compiling with `-Wall -Wextra` helps catch these mistakes.

`while` Loops

A `while` loop repeats a block of code as long as a condition is true. The condition is tested *before* each iteration, so if the condition is false from the start, the body never executes:

```
int countdown = 5;

while (countdown > 0) {
    std::cout << countdown << "... ";
    countdown--;
}
std::cout << "Vamos!\n";
// 5... 4... 3... 2... 1... Vamos!
```

Be careful to make sure the condition will eventually become false. If it never does, you have an infinite loop and your program will run forever (or until you press Ctrl+C).

Trap: Forgetting to update the loop variable is a classic source of infinite loops:

```
int i = 0;
while (i < 10) {
    std::cout << i << "\n";
    // oops, forgot i++ --- this runs forever
}
```

`do-while` Loops

A `do-while` loop is similar to `while`, but it tests the condition *after* the body. This guarantees the body executes at least once:

```
#include <iostream>
#include <string>

int main() {
    std::string input;

    do {
        std::cout << "Dime algo (or 'quit'): ";
        std::getline(std::cin, input);
        std::cout << "You said: " << input << "\n";
    } while (input != "quit");

    std::cout << "Adios!\n";
    return 0;
}
```

Notice the semicolon after `while (input != "quit")` — it is required for `do-while` and forgetting it is a syntax error.

Tip: Use `do-while` when the loop body must execute at least once. Menu loops and input validation are classic use cases. If you find yourself duplicating code before a `while` loop just to set up the first test, a `do-while` is probably cleaner.

`break` and `continue`

Two keywords let you alter the normal flow inside a loop.

`break` exits the nearest enclosing loop immediately. `continue` skips the rest of the current iteration and jumps to the next iteration. In a `while` or `do-while` loop, that means the condition test. In a `for` loop, it jumps to the update expression (e.g., `i++`) first, and then the condition is tested.

```
#include <iostream>
#include <string>

int main() {
    // break example: stop searching when we find what we want
    std::string tracks[] = {
        "Losing My Religion",
        "Bitter Sweet Symphony",
        "Zombie"
    };

    for (int i = 0; i < 3; i++) {
        if (tracks[i] == "Zombie") {
            std::cout << "Found it at index " << i << "\n";
            break;
        }
    }

    // continue example: skip even numbers
```

```
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0)
            continue;
        std::cout << i << " ";
    }
    std::cout << "\n";
    // 1 3 5 7 9

    return 0;
}
```

`break` and `continue` only affect the innermost loop they appear in. If you have nested loops, a `break` in the inner loop does not exit the outer loop.

`for` Loops

The `for` loop is the most common loop in C++. It packs the initialization, condition, and update into a single line:

```
for (init; condition; update) {
    // body
}
```

Here is a classic example:

```
for (int i = 0; i < 5; i++) {
    std::cout << i << " ";
}
std::cout << "\n";
// 0 1 2 3 4
```

The three parts of the `for` header work like this:

1. **init** runs once before the loop starts.
2. **condition** is tested before each iteration. If false, the loop ends.
3. **update** runs after each iteration, before the next condition test.

You can iterate over an array with a `for` loop using an index:

```
int scores[] = {90, 84, 77, 95, 88};
int n = sizeof(scores) / sizeof(scores[0]);

for (int i = 0; i < n; i++) {
    std::cout << "Score " << (i + 1) << ": " << scores[i] << "\n";
}
```

Any part of the `for` header can be omitted. Omitting all three creates an infinite loop:

```
for (;;) {
    // runs forever --- use break to exit
}
```

Tip: Declare loop variables inside the `for` statement when possible: `for (int i = 0; ...)`. This limits the variable's scope to the loop body, preventing accidental use after the loop ends.

Range-Based `for` Loop

C++ provides a more convenient way to loop over collections called the range-based `for` loop:

```
int scores[] = {90, 84, 77, 95, 88};

for (int s : scores) {
    std::cout << s << " ";
}
std::cout << "\n";
// 90 84 77 95 88
```

The variable `s` takes on each value in `scores` one at a time. You do not need to manage an index variable or worry about going out of bounds.

This is just a brief introduction. You will use range-based `for` loops extensively in the Containers chapter, where they really shine with `std::vector` and `std::array`.

Wut: The range-based `for` loop makes a *copy* of each element by default. If you want to modify the elements in place, or avoid copying large objects, use a reference: `for (int &s : scores)`. We will cover references in more detail in the Functions chapter.

`switch` Statements

A `switch` statement selects among multiple cases based on the value of an integer or enumeration expression. It is useful when you are comparing one variable against several specific values:

```
#include <iostream>

int main() {
    int track = 2;

    switch (track) {
    case 1:
        std::cout << "Losing My Religion\n";
        break;
    case 2:
        std::cout << "Bitter Sweet Symphony\n";
        break;
    case 3:
        std::cout << "Zombie\n";
        break;
    default:
        std::cout << "Unknown track\n";
        break;
    }

    return 0;
}
```

Each `case` label must be a compile-time constant — you cannot use variables or strings as case labels.

Fall-Through

The most important thing to understand about `switch` is **fall-through** behavior. If you forget a `break`, execution continues into the next case. Sometimes this is intentional:

```cpp
char grade = 'B';

switch (grade) {
case 'A':
case 'B':
case 'C':
    std::cout << "Passing\n";
    break;
case 'D':
case 'F':
    std::cout << "Not passing\n";
    break;
default:
    std::cout << "Invalid grade\n";
    break;
}
```

Here, cases `'A'`, `'B'`, and `'C'` all fall through to the same output. This is a common and useful pattern.

But accidental fall-through is a frequent bug:

```cpp
switch (x) {
case 1:
    std::cout << "uno\n";
    // oops, forgot break --- falls into case 2
case 2:
    std::cout << "dos\n";
    break;
}
```

If `x` is `1`, this prints both "uno" and "dos."

Trap: Every `case` should end with `break` unless you intentionally want fall-through. When you do use fall-through on purpose, add a comment like `// fall through` so the next person reading the code knows it is deliberate. C++17 introduced the `[[fallthrough]]` attribute to make this intent explicit and silence compiler warnings.

The `default` Case

The `default` case runs when no other case matches. It is optional, but good practice to always include one — it catches unexpected values and makes your intent clear.

Try It: Control Flow Starter

```cpp
#include <iostream>
#include <string>

int main() {
    // if / else
    int volume = 11;
    if (volume > 10) {
        std::cout << "It's a bitter sweet symphony, this life\n";
    } else {
        std::cout << "Turn it up\n";
    }
```

```cpp
// while loop
int n = 5;
while (n > 0) {
    std::cout << n << " ";
    n--;
}
std::cout << "Go!\n";

// do-while: keep asking until they say the magic word
std::string answer;
do {
    std::cout << "What's in your head? ";
    std::getline(std::cin, answer);
} while (answer != "zombie");
std::cout << "Zombie, zombie, zombie-ie-ie\n";

// for with break and continue
std::cout << "Odd numbers under 20: ";
for (int i = 1; i <= 100; i++) {
    if (i >= 20)
        break;
    if (i % 2 == 0)
        continue;
    std::cout << i << " ";
}
std::cout << "\n";

// range-based for
std::string playlist[] = {
    "Losing My Religion",
    "Bitter Sweet Symphony",
    "Zombie"
};
for (const std::string &song : playlist) {
    std::cout << "Now playing: " << song << "\n";
}

// switch
int choice = 2;
switch (choice) {
case 1:
    std::cout << "R.E.M.\n";
    break;
case 2:
    std::cout << "The Verve\n";
    break;
case 3:
    std::cout << "The Cranberries\n";
    break;
default:
    std::cout << "Unknown band\n";
    break;
}
```

```
    return 0;
}
```

Key Points

- `if`, `else if`, and `else` let your program make decisions based on conditions.
- `while` loops test the condition before each iteration. If the condition is false initially, the body never runs.
- `do-while` loops test the condition after each iteration, guaranteeing at least one execution of the body.
- `break` exits the nearest loop or `switch`. `continue` skips to the next iteration.
- `for` loops combine initialization, condition, and update in one line. Declare loop variables in the `for` header to limit their scope.
- Range-based `for` loops provide a cleaner way to iterate over arrays and containers.
- `switch` selects among cases using an integer or enum value. Watch for accidental fall-through — always use `break` unless fall-through is intentional.

Exercises

1. **Think about it:** When would you choose a `do-while` loop over a `while` loop? Describe a scenario where `do-while` is clearly the better choice and explain why.
2. **What does this print?**

   ```
   for (int i = 0; i < 5; i++) {
       if (i == 3)
           continue;
       std::cout << i << " ";
   }
   std::cout << "\n";
   ```

3. **What does this print?**

   ```
   int x = 2;
   switch (x) {
   case 1:
       std::cout << "uno ";
   case 2:
       std::cout << "dos ";
   case 3:
       std::cout << "tres ";
       break;
   default:
       std::cout << "other ";
   }
   std::cout << "\n";
   ```

4. **Where is the bug?**

   ```
   int i;
   int total = 0;
   for (i = 0; i < 10; i++);
   {
       total += i;
   }
   std::cout << "Total: " << total << "\n";
   ```

5. **Calculation:** How many times does the body of this loop execute?

```
int count = 0;
int i = 10;
do {
    count++;
    i--;
} while (i > 10);
```

6. **What does this print?**

```
for (int i = 1; i <= 20; i++) {
    if (i % 3 == 0 && i % 5 == 0) {
        std::cout << "both ";
    } else if (i % 3 == 0) {
        std::cout << "tres ";
    } else if (i % 5 == 0) {
        std::cout << "cinco ";
    }
}
std::cout << "\n";
```

7. **Where is the bug?**

```
int n = 0;
while (n != 10) {
    std::cout << n << " ";
    n += 3;
}
```

8. **Write a program** that asks the user for a number between 1 and 7 and prints the day of the week using a `switch` statement. If the number is out of range, print an error message. Use a `do-while` loop to keep asking until the user enters `0` to quit.

9. **What does this print?**

```
#include <iostream>

int main()
{
    for (int row = 1; row <= 3; ++row) {
        for (int col = 1; col <= row; ++col) {
            std::cout << "*";
        }
        std::cout << "\n";
    }
    return 0;
}
```

Then change the inner loop to `for (int col = 1; col <= 4 - row; ++col)` and predict the new output.

10. **What does this print?**

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<std::string> tracks = {"Wonderwall", "Creep", "Linger"};
```

```
    for (const std::string &track : tracks) {
        std::cout << "- " << track << "\n";
    }
    return 0;
}
```

Why does the loop variable use `const std::string &` instead of just `std::string`?

11. **Write a program** that uses `break` to find the first negative number in an array. Use the array `int values[] = {3, 7, 2, -5, 4, -1};`. Print the index and value of the first negative number you find. If no negative number is found, print `"none"`.

12. **Think about it:** When would you intentionally let a `switch` case fall through into the next case, and how do you tell the compiler the fall-through is intentional rather than an accidental missing `break`?