Gorgo C for C++ Programmers

April 11, 2026

5. Control Flow

Control flow in C will feel very familiar if you are coming from C++. The `if`, `while`, `for`, and `switch` statements work mostly the same way. The differences are small but worth knowing: C has no range-based `for` loops, no structured bindings, and no `std::optional`. In C89, all variable declarations had to appear at the top of a block before any statements — C99 relaxed this and let you declare variables anywhere, which is how you are used to writing code in C++.

The biggest conceptual difference is that C has no native `bool` type. Conditions are just integers: zero is false, and any nonzero value is true.

`if` / `else`

The `if` statement in C is identical to C++:

```c
int score = 85;

if (score >= 90) {
    printf("Excelente!\n");
} else if (score >= 70) {
    printf("Passing\n");
} else {
    printf("Try again\n");
}
```

The condition in an `if` is an integer expression. Zero means false, nonzero means true. There is no built-in `bool` type in C89. C99 added `_Bool` and the convenience header `<stdbool.h>`, which defines `bool`, `true`, and `false`:

```c
#include <stdbool.h>

bool is_hurricane = true;
if (is_hurricane) {
    printf("Here I am\n");
}
```

Without `<stdbool.h>`, C programmers traditionally use `int` for boolean values and `0/1` for false/true, or define their own macros.

Trap: Because conditions are just integers, assignments inside `if` are legal and a common source of bugs:

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

The compiler may warn you about this, but it will not stop you. If you mean to compare, use `==`. Compiling with `-Wall` helps catch these.

`while` and `do-while`

A `while` loop tests the condition *before* executing the body. If the condition is false from the start, the body never executes:

```c
int countdown = 5;
while (countdown > 0) {
    printf("%d... ", countdown);
    countdown--;
}
printf("Vamos!\n");
// 5... 4... 3... 2... 1... Vamos!
```

A `do-while` loop tests *after* the body, guaranteeing at least one iteration. This is useful when you need to perform an action before you can check whether to continue:

```c
#include <stdio.h>

int main(void) {
    int choice;
    do {
        printf("1) Rock  2) Paper  3) Scissors  0) Quit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
        printf("You picked %d\n", choice);
    } while (choice != 0);

    printf("Adios!\n");
    return 0;
}
```

The semicolon after `while (choice != 0)` is required — forgetting it is a syntax error.

Tip: Use `do-while` when the loop body must execute at least once. Menu loops and input validation loops 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`

`break` exits the nearest enclosing loop (or `switch`) immediately. `continue` skips the rest of the current iteration and jumps to the next one.

```c
#include <stdio.h>

int main(void) {
    /* break example: stop at the first multiple of 7 */
    for (int i = 1; i <= 20; i++) {
        if (i % 7 == 0) {
            printf("Found it: %d\n", i);
            break;
        }
    }

    /* continue example: skip odd numbers */
    for (int i = 0; i < 10; i++) {
        if (i % 2 != 0)
            continue;
        printf("%d ", i);
    }
    printf("\n");
    // 0 2 4 6 8

    return 0;
}
```

`break` and `continue` only affect the innermost loop. If you have nested loops and need to break out of an outer loop, you either use a flag variable or, in some cases, `goto` (discussed later in this chapter).

`for` Loops

The `for` loop has the same structure as in C++:

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

Here is a classic example iterating over an array:

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

for (int i = 0; i < n; i++) {
    printf("Score %d: %d\n", i + 1, scores[i]);
}
```

C99 allows you to declare the loop variable inside the `for` statement, just like modern C++. In C89, you had to declare it before the loop:

```c
/* C89 style */
int i;
for (i = 0; i < n; i++) {
    printf("%d\n", scores[i]);
}

/* C99 style — preferred */
```

```c
for (int i = 0; i < n; i++) {
    printf("%d\n", scores[i]);
}
```

Tip: C has no range-based `for` loop. There is no `for (auto x : vec)`. You always iterate with an index or a pointer. The `sizeof(arr) / sizeof(arr[0])` idiom gives you the element count of a stack-allocated array, but it does not work on pointers — a pointer does not carry size information. This is why C functions that take arrays almost always take a separate `size` parameter.

You can iterate over an array with a pointer instead of an index. This is idiomatic C and worth getting comfortable with:

```c
int nums[] = {10, 20, 30, 40, 50};
int *end = nums + sizeof(nums) / sizeof(nums[0]);

for (int *p = nums; p < end; p++) {
    printf("%d ", *p);
}
printf("\n");
// 10 20 30 40 50
```

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

```c
for (;;) {
    /* runs forever — use break to exit */
}
```

`switch` Statements

A `switch` statement selects among multiple cases based on an integer expression. If you have used `switch` in C++, the syntax is identical:

```c
#include <stdio.h>

int main(void) {
    int wind = 5;   /* Beaufort scale */

    switch (wind) {
    case 0:
        printf("Calm\n");
        break;
    case 5:
        printf("Fresh breeze\n");
        break;
    case 12:
        printf("Huracan!\n");
        break;
    default:
        printf("Wind level: %d\n", wind);
        break;
    }

    return 0;
}
```

Each `case` must be an integer constant expression. You cannot use strings, floats, or variables as case labels — only values the compiler can evaluate at compile time.

Fall-through is the most important thing to understand about `switch` in C. If you forget a `break`, execution falls through to the next case. This is sometimes intentional:

```c
char grade = 'B';

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

Here, cases `'A'`, `'B'`, and `'C'` all fall through to the same `printf`. This is deliberate and a common pattern. But accidental fall-through is a frequent bug:

```c
switch (x) {
case 1:
    printf("one\n");
    /* oops, forgot break — falls into case 2 */
case 2:
    printf("two\n");
    break;
}
```

If `x` is `1`, this prints both "one" and "two."

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. Some compilers recognize this comment and suppress warnings.

`goto`

In C++, you were probably taught to never use `goto`. In C, `goto` has a legitimate and widely used role: the **cleanup pattern**. When a function acquires multiple resources (files, memory, locks), and something goes wrong partway through, `goto` provides a clean way to release everything in the correct order.

Here is a brief example:

```c
#include <stdio.h>
#include <stdlib.h>

int process(const char *path) {
    int status = -1;

    FILE *f = fopen(path, "r");
```

```
    if (!f) return -1;

    char *buf = malloc(1024);
    if (!buf) goto close_file;

    /* ... do work with f and buf ... */
    status = 0;

    free(buf);
close_file:
    fclose(f);
    return status;
}
```

If `malloc` fails, control jumps to `close_file`, which closes the file that was already opened. Without `goto`, you would need deeply nested `if` statements or duplicated cleanup code. The `goto` cleanup pattern is used extensively in production C code, including the Linux kernel.

Tip: The `goto` cleanup pattern is covered in more detail in the Odds and Ends chapter. For now, just know that `goto` in C is not the taboo it is in C++. When used strictly for forward jumps to cleanup labels at the end of a function, it makes resource management clearer, not messier.

`goto` has two restrictions: you can only jump within the same function, and you cannot jump over a variable declaration that has an initializer (in C99+).

Try It: Control Flow Starter

```
#include <stdio.h>

int main(void) {
    /* if / else */
    int wind = 74;   /* mph */
    if (wind >= 74) {
        printf("Rock you like a hurricane!\n");
    } else if (wind >= 39) {
        printf("Tropical storm\n");
    } else {
        printf("Calm seas\n");
    }

    /* while */
    int n = 1;
    while (n <= 5) {
        printf("%d ", n);
        n++;
    }
    printf("\n");

    /* do-while: repeat until valid input */
    int guess;
    do {
        printf("Guess (1-10): ");
        scanf("%d", &guess);
    } while (guess < 1 || guess > 10);
```

```c
    printf("You guessed %d\n", guess);

    /* for with break and continue */
    printf("Even numbers up to 20: ");
    for (int i = 1; i <= 100; i++) {
        if (i > 20)
            break;
        if (i % 2 != 0)
            continue;
        printf("%d ", i);
    }
    printf("\n");

    /* switch */
    int track = 3;
    printf("Side B, Track %d: ", track);
    switch (track) {
    case 1:  printf("Big City Nights\n");   break;
    case 2:  printf("Wind of Change\n"); break;
    case 3:  printf("Rock You Like a Hurricane\n"); break;
    default: printf("Unknown track\n");     break;
    }

    return 0;
}
```

Key Points

- C's control flow statements (`if`, `while`, `do-while`, `for`, `switch`) are syntactically identical to C++.
- C uses integers for boolean conditions: 0 is false, nonzero is true. Include `<stdbool.h>` for `bool`, `true`, and `false` (C99+).
- `break` exits the nearest loop or `switch`. `continue` skips to the next iteration.
- C has no range-based `for` loop. Use index or pointer iteration.
- `switch` cases must be integer constants. Watch for accidental fall-through.
- `goto` is legitimate in C for the cleanup pattern — forward jumps to release resources in reverse order.

Exercises

1. **Think about it:** C uses `0` for false and nonzero for true, while C++ has a built-in `bool` type. What practical problems can arise from using integers as booleans? Can you think of a case where a nonzero value that is not `1` might cause a subtle bug?
2. **What does this print?**

   ```c
   for (int i = 0; i < 5; i++) {
       if (i == 3)
           continue;
       printf("%d ", i);
   }
   printf("\n");
   ```

3. **What does this print?**

   ```c
   int x = 2;
   switch (x) {
   case 1:
   ```

```
    printf("uno ");
case 2:
    printf("dos ");
case 3:
    printf("tres ");
    break;
default:
    printf("other ");
}
printf("\n");
```

4. **Where is the bug?**

```
int total = 0;
int i;
for (i = 0; i < 10; i++);
{
    total += i;
}
printf("Total: %d\n", total);
```

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. **Where is the bug?**

```
int level = 5;
if (level = 10) {
    printf("Max level!\n");
}
```

7. **Write a program** that reads integers from the user (using `scanf`) until the user enters `0`. Print the sum and average of all numbers entered (not counting the `0`). Use a `do-while` or `while` loop.