Gorgo C for C++ Programmers

April 11, 2026

4. Expressions

C and C++ share most of their operators, and if you have been writing C++ you will find the syntax immediately familiar. But there are important differences. C has no operator overloading — `+` always means arithmetic addition, never something a class author decided it should mean. The `<<` and `>>` operators are strictly bitwise shifts, not I/O operations. And C uses `int` for boolean results — there is no built-in `bool` type (though C99 added `_Bool` and `<stdbool.h>`).

This chapter walks through the operators you will use every day in C, highlights a few traps, and ends with the precedence table you will want to bookmark.

Assignment

The `=` operator assigns a value to a variable. In C, assignment is an **expression** — it produces a value, which is the value being assigned. This lets you chain assignments:

```
int a, b, c;
a = b = c = 0;   // all three are now 0
```

The chain works right to left: `c` gets `0`, then `b` gets the value of that assignment (also `0`), then `a` gets the same.

Because assignment is an expression, you can (and sometimes will) use it inside other expressions. A common pattern is assigning and testing a return value in one step:

```
int ch;
while ((ch = getchar()) != EOF) {
    putchar(ch);
}
```

Trap: Because `=` is assignment and `==` is comparison, a common mistake is writing `if (x = 5)` when you mean `if (x == 5)`. The first assigns 5 to `x` and then evaluates as true (since 5 is nonzero). Modern compilers warn about this, but it is still one of the most famous bugs in C:

```
int x = 0;
if (x = 5) {
    printf("This always runs!\n");  // x is now 5, which is true
}
```

Some programmers write the constant on the left — `if (5 == x)` — so that `if (5 = x)` would be a compiler error. This is called a **Yoda condition**.

Arithmetic Operators

The arithmetic operators work on numeric types just like in C++:

Operator	Operation	Example	Result
`+`	addition	`3 + 4`	`7`
`-`	subtraction	`10 - 3`	`7`
`*`	multiplication	`6 * 7`	`42`
`/`	division	`17 / 5`	`3`
`%`	remainder	`17 % 5`	`2`

Integer division truncates toward zero. This means `17 / 5` gives `3`, not `3.4`. If you want a floating-point result, at least one operand must be a floating-point type:

```
printf("%d\n", 17 / 5);       // 3
printf("%f\n", 17.0 / 5);     // 3.400000
```

The `%` operator gives the **remainder** after integer division. In C99 and later, the result of `%` has the same sign as the dividend (the left operand):

```
printf("%d\n",  17 %  5);   //  2
printf("%d\n", -17 %  5);   // -2
printf("%d\n",  17 % -5);   //  2
printf("%d\n", -17 % -5);   // -2
```

Wut: The `%` operator is often called "modulo," but it is technically the **remainder** operator. For positive numbers, remainder and modulo are the same. For negative numbers, they differ. In mathematics, modulo always returns a non-negative result. In C, `%` preserves the sign of the dividend. If you need a true modulo that always returns a non-negative value, you need to adjust the result yourself:

```
/* assumes m > 0 */
int mod(int a, int m) {
    int r = a % m;
    return r < 0 ? r + m : r;
}
```

Comparison and Logical Operators

Comparison operators produce `1` for true and `0` for false. The result type is `int`, not `bool`:

Operator	Meaning
`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

Logical operators combine boolean expressions:

Operator	Meaning
`&&`	logical AND
`||`	logical OR
`!`	logical NOT

Both `&&` and `||` use **short-circuit evaluation**, just like C++. With `&&`, if the left side is false, the right side is never evaluated. With `||`, if the left side is true, the right side is skipped:

```
int *p = NULL;
if (p != NULL && *p > 0) {
    // safe — *p is only evaluated if p is not NULL
}
```

No Built-in `bool`

In C++, `bool` is a built-in type. In C89, there is no boolean type at all — you use `int` where `0` is false and anything nonzero is true. C99 added `_Bool` as a keyword and `<stdbool.h>` as a convenience header that defines `bool`, `true`, and `false`:

```
#include <stdbool.h>

bool is_valid = true;
if (is_valid) {
    printf("All aboard the Crazy Train!\n");
}
```

Without `<stdbool.h>`, you will see code like this:

```
int done = 0;    // 0 means false
while (!done) {
    // ... do work ...
    done = 1;    // nonzero means true
}
```

Tip: In C, any nonzero value is true. The number `42`, the character `'A'`, and the pointer `0x7fff` are all true. Only `0` (and `NULL` for pointers) is false. This is why you can write `if (ptr)` instead of `if (ptr != NULL)` — they mean the same thing.

Bitwise Operators

Bitwise operators work on the individual bits of integer values. In C++, `<<` and `>>` are commonly used for stream I/O. In C, they are exclusively bit shift operators.

Operator	Operation	Example	Result
`&`	bitwise AND	`0xF0 & 0x3C`	`0x30`
`|`	bitwise OR	`0xF0 | 0x0F`	`0xFF`
`^`	bitwise XOR	`0xFF ^ 0x0F`	`0xF0`
`~`	bitwise NOT	`~0x00`	`0xFF...FF`
`<<`	left shift	`1 << 3`	`8`
`>>`	right shift	`16 >> 2`	`4`

Flag Manipulation

One of the most common uses of bitwise operators in C is manipulating **flags** — individual bits within an integer that each represent an on/off setting:

```c
#include <stdio.h>

#define FLAG_READ    (1 << 0)   // bit 0: 0x01
#define FLAG_WRITE   (1 << 1)   // bit 1: 0x02
#define FLAG_EXEC    (1 << 2)   // bit 2: 0x04

int main(void) {
    unsigned int perms = 0;

    // Set bits with |
    perms |= FLAG_READ;
    perms |= FLAG_WRITE;
    printf("perms = 0x%02X\n", perms);   // 0x03

    // Check a bit with &
    if (perms & FLAG_READ) {
        printf("Read permission is set\n");
    }
    if (!(perms & FLAG_EXEC)) {
        printf("Execute permission is NOT set\n");
    }

    // Toggle a bit with ^
    perms ^= FLAG_WRITE;
    printf("After toggling write: 0x%02X\n", perms);   // 0x01

    // Clear a bit with & and ~
    perms &= ~FLAG_READ;
    printf("After clearing read: 0x%02X\n", perms);    // 0x00

    return 0;
}
```

The pattern is straightforward:

- **Set** a bit: `flags |= BIT;`
- **Clear** a bit: `flags &= ~BIT;`
- **Toggle** a bit: `flags ^= BIT;`
- **Check** a bit: `if (flags & BIT)`

Tip: Shifting `1` to create bit masks — `(1 << n)` — is a common idiom in C for hardware registers, permission flags, and option bitmasks. It is clearer than writing raw hex values because you can see exactly which bit position you are targeting.

Compound Assignment Operators

Compound assignment operators combine an arithmetic or bitwise operation with assignment. They work exactly as in C++:

Operator	Equivalent to
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a %= b`	`a = a % b`
`a &= b`	`a = a & b`
`a |= b`	`a = a | b`
`a ^= b`	`a = a ^ b`
`a <<= b`	`a = a << b`
`a >>= b`	`a = a >> b`

These are not just shortcuts — they express intent more clearly. When you write `count += 1`, the reader knows you are incrementing `count`. When you write `count = count + 1`, the reader has to verify that the same variable appears on both sides.

Increment and Decrement

The `++` and `--` operators increment or decrement a variable by one. They come in prefix and postfix forms:

```c
int x = 5;
int a = ++x;   // prefix: x becomes 6, then a gets 6
int b = x++;   // postfix: b gets 6 (current value), then x becomes 7
```

In a standalone statement, `x++` and `++x` do the same thing — increment `x`. The difference only matters when the result is used in a larger expression.

Trap: Do not modify a variable more than once in the same expression. The result is **undefined behavior**:

```c
int i = 3;
int result = i++ + ++i;   // UNDEFINED BEHAVIOR — do not do this
```

The compiler is free to evaluate the sub-expressions in any order, and different compilers (or even the same compiler with different optimization levels) may produce different results. If you need multiple modifications, use separate statements.

The Ternary Operator

The ternary operator `? :` is a compact alternative to `if/else` for simple value selection:

```c
int volume = 11;
const char *verdict = (volume > 10) ? "Loco" : "Tranquilo";
printf("Volume %d: %s\n", volume, verdict);   // Volume 11: Loco
```

The syntax is `condition ? value_if_true : value_if_false`. The ternary operator is an expression, so it produces a value that can be used in assignments, function arguments, or anywhere a value is expected:

```
printf("Track %d is %s\n", track,
       (track % 2 == 0) ? "even" : "odd");
```

Tip: The ternary operator is great for simple one-line decisions. If your condition or either branch is complex, use a regular `if/else` instead. Readability matters more than cleverness.

Operator Precedence

When multiple operators appear in an expression, C evaluates them according to a precedence table. Here are the most important levels, from highest (evaluated first) to lowest:

Precedence	Operators	Description
1	`() [] -> .`	grouping, subscript, member access
2	`! ~ ++ -- + - * & (type) sizeof`	unary operators
3	`* / %`	multiplication, division, remainder
4	`+ -`	addition, subtraction
5	`<< >>`	bitwise shifts
6	`< <= > >=`	relational
7	`== !=`	equality
8	`&`	bitwise AND
9	`^`	bitwise XOR
10	`|`	bitwise OR
11	`&&`	logical AND
12	`||`	logical OR
13	`? :`	ternary
14	`= += -=` etc.	assignment
15	`,`	comma

Common Precedence Traps

The most dangerous precedence surprise is that **bitwise operators bind more loosely than comparison operators**:

```
// WRONG — this checks (x) & (0x04 == 0x04), which is (x) & (1)
if (x & 0x04 == 0x04) { ... }

// RIGHT — parentheses fix the precedence
if ((x & 0x04) == 0x04) { ... }
```

Similarly, `||` has lower precedence than `&&`, which matches mathematical convention (AND before OR) but can surprise you:

```
// This evaluates as: a || (b && c)  — not (a || b) && c
if (a || b && c) { ... }
```

Tip: When in doubt, use parentheses. No one will fault you for writing `(a & b) == c` instead of relying on precedence rules. The few extra characters make your intent unmistakable and save the next reader (who might be you) from having to look up the precedence table.

Try It: Expressions Starter

```c
#include <stdio.h>

int main(void) {
    // Assignment chaining
    int a, b, c;
    a = b = c = 1980;
    printf("a=%d b=%d c=%d\n", a, b, c);

    // Integer division and remainder
    printf("17 / 5 = %d\n", 17 / 5);
    printf("17 %% 5 = %d\n", 17 % 5);
    printf("-17 %% 5 = %d\n", -17 % 5);

    // Boolean values are just ints
    printf("(10 > 5) = %d\n", 10 > 5);
    printf("(10 < 5) = %d\n", 10 < 5);

    // Bitwise flag manipulation
    unsigned int flags = 0;
    flags |= (1 << 0);   // set bit 0
    flags |= (1 << 2);   // set bit 2
    printf("flags = 0x%02X\n", flags);        // 0x05
    printf("bit 1 set? %d\n", (flags & (1 << 1)) != 0);  // 0
    printf("bit 2 set? %d\n", (flags & (1 << 2)) != 0);  // 1

    // Ternary operator
    int vol = 11;
    printf("Volume: %s\n", (vol > 10) ? "Muy alto" : "Normal");

    // Compound assignment
    int total = 100;
    total += 50;
    total -= 20;
    total *= 2;
    printf("total = %d\n", total);   // 260

    return 0;
}
```

Key Points

- Assignment is an expression in C — it produces a value, enabling chaining (`a = b = c = 0`) and assignment within conditions.
- Integer division truncates toward zero. The `%` operator gives the remainder, which preserves the sign of the dividend.
- C uses `int` for boolean results: `0` is false, nonzero is true. Include `<stdbool.h>` for `bool`, `true`, and `false`.
- Bitwise `<<` and `>>` are shifts only — they are not overloaded for I/O as in C++.
- Use `|` to set bits, `&` to check bits, `^` to toggle bits, and `& ~` to clear bits.
- Bitwise operators have lower precedence than comparison operators. Always use parentheses when mixing them.
- Never modify a variable more than once in the same expression — the result is undefined behavior.

Exercises

1. **Think about it:** In C++, you can overload operators to give `+`, `<<`, `==`, and others custom meanings for your classes. C does not allow operator overloading. What advantage does this give you when reading unfamiliar C code? Can you think of a situation where operator overloading would have been genuinely useful in C?

2. **What does this print?**

   ```c
   int x = 10;
   int y = x++ + ++x;
   printf("%d %d\n", x, y);
   ```

   (Be careful — is the answer even defined?)

3. **Calculation:** What is the result of each of these expressions?

   ```c
   25 / 4
   25 % 4
   -25 % 4
   (1 << 4) | (1 << 1)
   0xFF & 0x0F
   ```

4. **Where is the bug?**

   ```c
   int status = 0x07;
   if (status & 0x04 == 0x04) {
       printf("Bit 2 is set\n");
   }
   ```

5. **What does this print?**

   ```c
   int a = 5, b = 10;
   a ^= b;
   b ^= a;
   a ^= b;
   printf("a=%d b=%d\n", a, b);
   ```

6. **Where is the bug?**

   ```c
   int count = 0;
   if (count = 0) {
       printf("El contador es cero\n");
   } else {
       printf("El contador no es cero\n");
   }
   ```

7. **Write a program** that takes an `unsigned int` and prints its value in binary (most significant bit first). Use bitwise operators to test each bit. Test it with the values `0`, `1`, `255`, and `1024`.