C++ Operators

February 28, 2026

Operators are symbols that take operands and produce a result. That definition isn't very satisfying though. The best way to understand operators is to look at ones you have already seen: =, +, <. All of these operators are **in-fix** operators: they occur between the things they are operating on. `a = 3` assigns the value 3 to `a`. `b+c` represents the sum of `b` and `c`. `a<b` returns true if `a` is less than `b`. You have also seen the **unary** operators: `++ --`. There is also the negation unary operator: `-a`, which returns the negation of `a`. You will see more, but you can think of operators as functions that are invoked with symbols rather than a function name like `sum(b,c)`. In fact, C++ often refers to operators as operator functions.

You have already used several operators without thinking much about them. In Chapter 1, you used `::` (the scope resolution operator) to call `std::println`. In Chapter 2, you used `=` to assign values to variables and `>>` to extract input from `std::cin`. You have compared values with `==` and `<`. This chapter takes a closer look at all the operators C++ offers, starting with the ones you know and building up to the ones you have not seen yet.

1. Math Operators

Assignment: `=`

You have used `=` since Chapter 2 to give values to variables:

```
int score = 100;
```

> **Tip:** Technically, `int score = 100;` is **initialization**, not assignment. You learned about initialization with `{}` in Chapter 2. Assignment happens when you change a variable that already exists, like `score = 200;`. The distinction matters more as you get deeper into C++, but for now just know they are not the same thing.

One useful property of `=` is that it returns the value it assigned. This lets you chain assignments together:

```
int a, b;
a = b = 3;  // b gets 3, then a gets 3
```

The chain is evaluated right-to-left: `b` is assigned `3` first, and that result (`3`) is then assigned to `a`.

Arithmetic: `+`, `-`, `*`, `/`

These work as you would expect from math class:

```
int x = 10 + 3;   // 13
int y = 10 - 3;   // 7
int z = 10 * 3;   // 30
int w = 10 / 3;   // 3, not 3.333!
```

Watch out for integer division. When both operands are integers, `/` discards the remainder. If you need a decimal result, use `double`:

```
double precise = 10.0 / 3.0;  // 3.33333...
```

Modulus: `%`

The modulus operator `%` gives you the remainder after division. It works only with integers.

```
int remainder = 10 % 3;  // 1, because 10 / 3 = 3 remainder 1
```

Modulus is surprisingly useful. A common use is checking whether a number is even or odd:

```
if (number % 2 == 0) {
    std::println("Even");
} else {
    std::println("Odd");
}
```

You can also use it for wrapping, like a clock. Hours wrap around after 12:

```
int hour = 15 % 12;  // 3
```

Or extracting digits from a number:

```
int last_digit = 1985 % 10;   // 5
int last_two   = 1985 % 100;  // 85
```

Increment and Decrement: `++`, `--`

You first saw `++` in Chapter 4, where it was used to count through arrays. Adding one to a variable is so common that C++ gives you a shortcut. In fact, the `++` in C++ comes from this operator.

There are two forms. **Prefix** increments first, then returns the new value:

```
int x = 5;
int y = ++x;  // x is now 6, y is 6
```

Postfix returns the current value first, then increments:

```
int x = 5;
int y = x++;  // y is 5, x is now 6
```

The decrement operator `--` works the same way but subtracts one.

> **Tip:** In loops and standalone statements, prefer prefix (`++i`) over postfix (`i++`). For integers the difference is negligible, but with iterators and other types, prefix can be more efficient because it avoids creating a temporary copy.

Compound Assignment: `+=`, `-=`, `*=`, `/=`, `%=`

When you want to update a variable based on its current value, compound assignment keeps your code concise:

```
int score = 100;
score += 10;   // same as: score = score + 10;  -> 110
score -= 25;   // same as: score = score - 25;  -> 85
score *= 2;    // same as: score = score * 2;   -> 170
score /= 5;    // same as: score = score / 5;   -> 34
score %= 10;   // same as: score = score % 10;  -> 4
```

These are especially handy for running totals and accumulators:

```
double total{};
for (const auto price : prices) {
    total += price;
}
```

A Note on Precedence

Multiplication, division, and modulus are evaluated before addition and subtraction, just like in math class. When in doubt, use parentheses to make your intent clear:

```
int result = 2 + 3 * 4;      // 14, not 20
int clear  = 2 + (3 * 4);    // 14, same thing but obvious
int other  = (2 + 3) * 4;    // 20
```

2. Comparison Operators

You have seen these since Chapter 2. Each comparison returns a `bool` — either `true` or `false`:

```
int a = 5, b = 10;
bool eq = (a == b);   // false  - equal to
```

```cpp
bool ne = (a != b);   // true   - not equal to
bool lt = (a < b);    // true   - less than
bool le = (a <= b);   // true   - less than or equal to
bool gt = (a > b);    // false  - greater than
bool ge = (a >= b);   // false  - greater than or equal to
```

These are straightforward, but there is one trap that catches nearly everyone at least once.

Tip: Do not confuse `=` (assignment) with `==` (comparison). The expression `if (x = 5)` does not check whether `x` equals `5`. It *assigns* `5` to `x`, and since `5` is nonzero (truthy), the condition is always true. This is a legal C++ statement, so the compiler may not warn you. Always double-check that you used `==` in conditions.

Tip: C++20 introduced the **three-way comparison operator `<=>`**, sometimes called the "spaceship operator." It compares two values and tells you whether the left side is less than, equal to, or greater than the right side — all in one operation. You can safely ignore it for now, but you may see it in modern C++ code.

3. Logical Operators

Logical operators combine or invert `bool` values. You use them to build more complex conditions.

AND: `&&`

Both conditions must be true:

```cpp
if (age >= 18 && has_ticket) {
    std::println("Bienvenido al concierto!");
}
```

OR: `||`

At least one condition must be true:

```cpp
if (day == "Saturday" || day == "Sunday") {
    std::println("Fin de semana");
}
```

NOT: `!`

Inverts a boolean value:

```cpp
if (!game_over) {
    std::println("Keep playing");
}
```

Because `!` is a unary operator at precedence level 3, it binds tighter than almost every other operator. This means `!` applies to the very next value, not to a larger expression. Watch out for this when negating comparisons:

```
int score = 100;

// Bug: !score evaluates first (0), then 0 > 80 is false
if (!score > 80) {
    std::println("You failed");  // never prints - even with score 0
}

// Correct: negate the whole comparison
if (!(score > 80)) {
    std::println("You failed");
}
```

In the buggy version, `!score` converts `score` to `bool` (`true` for any non-zero value), negates it to `false` (which is `0`), and then compares `0 > 80`. The result is always `false` regardless of `score`. Parentheses fix this by letting `>` evaluate first.

Short-Circuit Evaluation

C++ evaluates logical expressions left-to-right and stops as soon as the result is determined. 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. This is useful for guarding against errors:

```
if (index < size && data[index] > 0) {
    // safe: data[index] is only accessed if index is valid
}
```

> **Tip:** Do not confuse `&&` and `||` (logical) with `&` and `|` (bitwise). The bitwise versions operate on individual bits and do *not* short-circuit. Using `&` when you meant `&&` can cause subtle bugs and unexpected behavior.

Precedence

`!` has the highest precedence of the three, followed by `&&`, then `||`. Use parentheses to make complex conditions readable:

```
// Without parentheses - && binds tighter than ||
if (is_vip || age >= 21 && has_id) { ... }

// Clearer with parentheses
if (is_vip || (age >= 21 && has_id)) { ... }
```

Tip: Some safety-critical coding standards, such as MISRA C++, require parentheses around every non-trivial sub-expression rather than relying on operator precedence. This eliminates an entire class of bugs caused by misremembering precedence rules.

4. The Ternary Operator: `?:`

Sometimes you need a simple either-or value. An `if/else` block works, but the ternary operator gives you a compact one-liner:

```
condition ? value_if_true : value_if_false
```

For example:

```
int score = 85;
std::string result = (score >= 60) ? "Passed" : "Failed";
std::println("{}", result);  // Passed
```

It is handy for quick assignments:

```
std::string greeting = (hora < 12) ? "Buenos dias" : "Buenas tardes";
```

Tip: Avoid nesting ternary operators. An expression like `a ? b : c ? d : e` is hard to read. Use `if/else` when the logic gets more complex than a single condition.

Note: The above tip is generally accepted best practice but Ben loves writing:
```
auto rc = a ? b :
          c ? d :
          e ? f :
          g;
```
He thinks it looks like a switch statement. We haven't been able to convince him that this is not beautiful code!

5. Bitwise Operators

Bitwise operators work on the individual bits of integer values. Before diving in, here is a quick refresher on binary. For a deeper look at binary, hexadecimal, two's complement, and how numbers are stored, see the Numbers chapter.

Binary Numbers

Computers store numbers in binary — base 2. Each digit is a **bit** that is either `0` or `1`. An 8-bit number looks like this:

```
Bit position:   7   6   5   4   3   2   1   0
Value:        128  64  32  16   8   4   2   1
```

For example, the number `42` in binary is `00101010`:

```
0*128 + 0*64 + 1*32 + 0*16 + 1*8 + 0*4 + 1*2 + 0*1 = 42
```

Bitwise AND: `&`

Each bit in the result is `1` only if both corresponding bits are `1`.

```
//   0b00101010  (42)
// & 0b00001111  (15)
// = 0b00001010  (10)
int masked = 42 & 0x0F;  // 10
```

Common use: masking — extracting specific bits from a value. You can also use `&` to check whether a particular bit is set:

```
if (flags & 0b00000100) {
    std::println("Bit 2 is set");
}
```

And combined with `~`, you can clear a bit:

```
flags &= ~0b00000100;  // clear bit 2
```

Bitwise OR: `|`

Each bit in the result is `1` if either corresponding bit is `1`.

```
//   0b00001010  (10)
// | 0b11000000  (192)
// = 0b11001010  (202)
int flags = 10 | 192;  // 202
```

Common use: setting bits, combining flags.

Bitwise XOR: `^`

Each bit in the result is `1` if the corresponding bits are *different*.

```
//   0b00101010  (42)
// ^ 0b00001111  (15)
// = 0b00100101  (37)
int toggled = 42 ^ 15;  // 37
```

Common use: toggling (flipping) specific bits.

Bitwise NOT: `~`

Inverts every bit. A `0` becomes `1` and a `1` becomes `0`.

```
unsigned char mask = 0b00001111;
unsigned char flipped = ~mask;  // 0b11110000 (240)
```

Note that `~` promotes its operand to `int` before flipping the bits. The result is then truncated back when stored in an `unsigned char`.

> **Tip:** Avoid `~` with signed types. The result depends on the size of the type and can produce negative values due to two's complement representation.

Left Shift: << and Right Shift: >>

Shifting moves bits left or right by a given number of positions.

```
int val = 1;
int shifted_left  = val << 3;  // 8  (1 * 2^3)
int shifted_right = 8 >> 2;    // 2  (8 / 2^2)
```

Left shifting by `n` is equivalent to multiplying by 2^n. Right shifting by `n` is equivalent to dividing by 2^n (for unsigned values).

Common use: efficient multiplication/division by powers of two, and positioning bits in flags or registers.

Compound Assignment Forms

Just like the arithmetic operators, each bitwise operator has a compound form:

```
int flags = 0b00000000;
flags |= 0b00000100;   // set bit 2        -> 0b00000100
flags &= 0b11111100;   // clear bits 0-1   -> 0b00000100
flags ^= 0b00000100;   // toggle bit 2     -> 0b00000000
flags <<= 2;            // shift left by 2
flags >>= 1;            // shift right by 1
```

> **Tip:** Bitwise operators have surprisingly low precedence — lower than `==`. The expression `x & 0xFF == 0` is parsed as `x & (0xFF == 0)`, which is almost certainly not what you want. Always use parentheses with bitwise operators: `(x & 0xFF) == 0`.

6. Other Operators

You will encounter a few more operators as you progress through the textbook. Here is a brief overview:

- `sizeof` — returns the size of a type or variable in bytes: `sizeof(int)`
- `()` — function call (Chapter 1) and grouping for precedence
- `[]` — array subscript (Chapter 4): `numbers[0]`
- `.` and `->` — member access (Chapter 10): `stock.price` or `ptr->price`
- `*` and `&` (unary) — dereference and address-of (Chapter 12)
- `,` — the comma operator evaluates two expressions and returns the second. You will rarely need it, and it can make code confusing. Avoid it.
- `::` — scope resolution (Chapter 1): `std::println`

Each of these will be covered in detail when you reach the relevant chapter.

7. Operator Overloading

Sometimes the same operator does different things depending on the types involved. This is called **operator overloading**, and you have been using it since Chapter 1.

The `+` operator adds numbers, but with `std::string` it concatenates:

```
int sum = 5 + 3;  // 8

std::string song = "Take On "s + "Me"s;  // "Take On Me"
```

The `<<` operator is a bitwise left shift for integers, but with `std::cout` it is the stream insertion operator:

```
int shifted = 1 << 4;           // 16 (bitwise shift)
std::cout << "Hola, mundo!\n";  // stream insertion
```

Similarly, `>>` is both bitwise right shift and the stream extraction operator you use with `std::cin`:

```
int half = 8 >> 1;         // 4 (bitwise shift)
std::cin >> number;         // stream extraction
```

In Chapter 15, you will see how to define `operator==` for your own custom types. For now, just be aware that an operator's meaning depends on what it is applied to.

> **Tip:** When you see an unfamiliar use of an operator, check what types the operands are. The type determines which version of the operator is called.

8. Precedence

When an expression has multiple operators, C++ uses **precedence** rules to decide which operator is evaluated first. Higher precedence means the operator is applied earlier. Here is the full table:

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`() [] . -> ++ --`	Function call, subscript, member access, postfix inc/dec, casts	Left-to-right

Precedence	Operator	Description	Associativity
3	`++ -- + - ! ~ * &`	Prefix inc/dec, unary plus/minus, logical NOT, bitwise NOT, deref, address-of	Right-to-left
5	`* / %`	Multiplication, division, modulus	Left-to-right
6	`+ -`	Addition, subtraction	Left-to-right
7	`<< >>`	Bitwise left and right shift	Left-to-right
8	`< <= > >=`	Relational operators	Left-to-right
9	`== !=`	Equality and inequality	Left-to-right
10	`&`	Bitwise AND	Left-to-right
11	`^`	Bitwise XOR	Left-to-right
12	`|`	Bitwise OR	Left-to-right
13	`&&`	Logical AND	Left-to-right
14	`||`	Logical OR	Left-to-right
15	`?: = += -= *= /= %= <<= >>= &= ^= |=`	Ternary, assignment, compound assignment	Right-to-left
16	`,`	Comma operator	Left-to-right

How to Read the Table

A **lower precedence number** means the operator binds more tightly. Operators at precedence 5 (`*`, `/`, `%`) are evaluated before operators at precedence 6 (`+`, `-`). That is why `2 + 3 * 4` equals `14`, not `20`.

Associativity tells you the direction when operators of the *same* precedence appear together. Most operators are left-to-right:

```
int val = 10 - 3 - 2;  // (10 - 3) - 2 = 5, not 10 - (3 - 2) = 9
```

Assignment operators are right-to-left, which is why chaining works:

```
a = b = 3;  // b = 3 first, then a = 3
```

Here is a quick example putting it together:

```cpp
int result = 2 + 3 * 4 > 10 && true;
// Step 1: 3 * 4 = 12        (precedence 5)
// Step 2: 2 + 12 = 14       (precedence 6)
// Step 3: 14 > 10 = true    (precedence 8)
// Step 4: true && true       (precedence 13)
// result = 1 (true)
```

Tip: You do not need to memorize this table. When you are unsure about the order, add parentheses. They make your intent obvious to both the compiler and anyone reading your code. A few extra `()` are far better than a subtle bug.

Conclusion

Operators are the building blocks of expressions in C++. Here are the key takeaways from this chapter:

- **Use `==` for comparison, not `=`.** This is the most common operator mistake in C++. Train yourself to double-check conditions.
- **Use parentheses when in doubt.** Precedence rules are hard to memorize, and `()` makes your intent clear to both the compiler and the next person reading your code.
- **Do not confuse logical and bitwise operators.** `&&` and `||` short-circuit and work with boolean logic. `&` and `|` operate on individual bits.
- **Operators can be overloaded.** The same symbol can mean different things depending on the types involved. When something looks unfamiliar, check the types.

You now have a solid understanding of how C++ operators work. As you continue through the textbook, you will see these operators everywhere — in loops, conditions, function calls, and class definitions. With practice, reading and writing expressions will become second nature. No te preocupes — you have got this. Nos vemos en el próximo capítulo!

Content outline and editorial support from Ben. Words by Claude, the Opus. Special thanks to Khalil Estell for thorough review and suggestions!