Gorgo Starting C++

April 11, 2026

Contents

4. Expressions

You know how to declare variables and store data (Chapter 2) and how to work with text (Chapter 3). But so far the only operation you have really used is assignment with `=` — storing a value and printing it back. With just assignment you cannot add two numbers, compare a score to a threshold, or check whether a string is empty. You can store data, but you cannot do anything with it. Expressions are how you tell C++ to compute, compare, and combine values. In this chapter you will learn the full set of operators C++ provides — arithmetic, logical, bitwise, and more — how they combine in expressions, and the precedence rules that determine the order of evaluation.

Assignment

The simplest operator is assignment with `=`. As you saw in Chapter 2, assignment stores a value in a variable.

```
int jumps = 0;
jumps = 42;
```

The right side is evaluated first, then the result is stored in the variable on the left. This means you can use the variable itself on the right side.

```
int count = 10;
count = count + 1;  // count is now 11
```

Trap: Do not confuse `=` (assignment) with `==` (equality comparison). Writing `if (x = 5)` assigns 5 to `x` instead of comparing `x` to 5. The compiler may warn you about this, but it is still valid C++.

Arithmetic Operators

C++ provides the standard math operators.

Operator	Operation	Example	Result
`+`	addition	`7 + 3`	`10`
`-`	subtraction	`7 - 3`	`4`
`*`	multiplication	`7 * 3`	`21`
`/`	division	`7 / 3`	`2`
`%`	modulo	`7 % 3`	`1`

These work the way you expect for the most part, but division has an important detail.

Integer Division

When both operands are integers, `/` performs integer division — it drops the fractional part.

```
int result = 7 / 3;  // 2, not 2.333...
```

The fractional part is simply discarded; it does not round. So `7 / 3` is `2` and `-7 / 3` is `-2`.

If you want the full decimal result, at least one operand must be a floating-point type.

```
double result = 7.0 / 3;   // 2.333...
double also   = 7 / 3.0;   // 2.333...
int nope      = 7 / 3;     // 2 --- both operands are int
```

Trap: Integer division by zero crashes your program. There is no exception, no error message — just a crash (or undefined behavior). Always check your divisor before dividing.

Modulo

The `%` operator gives the remainder after integer division.

```
int leftover = 7 % 3;   // 1, because 7 = 2*3 + 1
int even = 10 % 2;      // 0, so 10 is even
```

Modulo only works with integers. You cannot use `%` with `float` or `double`.

A common use for modulo is checking if a number is even or odd.

```
if (number % 2 == 0) {
    std::cout << "even" << std::endl;
} else {
    std::cout << "odd" << std::endl;
}
```

Comparison Operators

Comparison operators compare two values and produce a `bool` result: `true` or `false`.

Operator	Meaning	Example	Result
`==`	equal to	`5 == 5`	`true`
`!=`	not equal to	`5 != 3`	`true`
`<`	less than	`3 < 5`	`true`
`>`	greater than	`3 > 5`	`false`
`<=`	less than or equal to	`5 <= 5`	`true`
`>=`	greater than or equal to	`3 >= 5`	`false`

As you saw in Chapter 3, these also work on strings, comparing them character by character.

Logical Operators

Logical operators combine boolean expressions.

Operator	Meaning	Example	Result
`&&`	AND	`true && false`	`false`
`||`	OR	`true || false`	`true`
`!`	NOT	`!true`	`false`

These are used to build more complex conditions. They work the same way they do in English. `A && B` is true if both `A` and `B` are true, otherwise it is false. `A || B` is true as long as `A` or `B` are true — this includes the case when both are true.

Let's look at this rule: you must have two pencils or one pen and not have a calculator to take a test. Unfortunately, in English, it is ambiguous whether or not you can take the test if you have a pencil and a calculator. Fortunately, we can make it clear in code using parentheses.

```cpp
int pencils = 2;
int pens = 1;
bool has_calculator = true;
bool can_take_test = (pencils == 2 || pens == 1) && !has_calculator;
```

In the above scenario, the student cannot take the test because they have a calculator. If we change `has_calculator` to `false`, `can_take_test` will become `true`.

Tip: When mixing logical operators use parentheses. This rule is worth repeating :) . There are rules about order of operation, but your average developer will not be certain that they remember them correctly. They make your intent clear to both the compiler and anyone reading your code. The exception to this rule is the `!` operator which will be evaluated before other logical operators.

Short-Circuit Evaluation

C++ evaluates logical operators left to right and stops as soon as the result is known.

With `&&`, if the left side is `false`, the right side is never evaluated — the result is already `false` no matter what. With `||`, if the left side is `true`, the right side is never evaluated — the result is already `true`.

```cpp
int x = 0;
if (x != 0 && 10 / x > 2) {
    // safe: if x is 0, the division never happens
}
```

This is not just an optimization — it is a guarantee you can rely on. The example above would crash without short-circuit evaluation because dividing by zero is undefined behavior.

Tip: Short-circuit evaluation lets you write guard conditions. Check that an operation is safe on the left side of `&&` before performing it on the right side.

Increment and Decrement

The `++` and `--` operators add or subtract 1 from a variable. They come in two forms: prefix and postfix.

```cpp
int n = 5;
++n;    // prefix: n is now 6
n++;    // postfix: n is now 7
--n;    // prefix: n is now 6
n--;    // postfix: n is now 5
```

When used as a standalone statement, prefix and postfix do the same thing. The difference appears when the result is used in a larger expression.

- **Prefix** (`++n`): increments `n`, then returns the new value.
- **Postfix** (`n++`): returns the current value of `n`, then increments it.

```cpp
int a = 5;
int b = ++a;   // a is 6, b is 6 (increment first, then use)
int c = a++;   // a is 7, c is 6 (use current value, then increment)
```

Tip: When you do not need the old value, prefer prefix `++n` out of habit. For built-in types the compiler optimizes them to be the same, but with more complex types (like iterators) prefix can be faster because it does not need to make a copy of the old value.

Compound Assignment Operators

Compound assignment operators combine an arithmetic or bitwise operation with assignment. Instead of writing `x = x + 5`, you can write `x += 5`.

Operator	Equivalent
`+=`	`x = x + y`
`-=`	`x = x - y`
`*=`	`x = x * y`
`/=`	`x = x / y`
`%=`	`x = x % y`
`&=`	`x = x & y`
`|=`	`x = x | y`
`^=`	`x = x ^ y`
`<<=`	`x = x << y`
`>>=`	`x = x >> y`

```
int score = 100;
score += 50;   // score is now 150
score -= 25;   // score is now 125
score *= 2;    // score is now 250
```

These are just shorthand. There is no difference in behavior between `score += 50` and `score = score + 50`.

Bitwise Operators

Bitwise operators work on the individual bits of integer values. These are not something you will use every day, but they are essential for systems programming, hardware interfaces, and flags. The Numbers chapter explains how to think about a number as a row of bits and works through what each of these operators does to those bits. For now, here is a quick preview of which symbols exist.

Operator	Operation
`&`	AND
`|`	OR
`^`	XOR
`~`	NOT (complement)
`<<`	left shift
`>>`	right shift

Trap: `&&` and `&` and `||` and `|` are very different operations. The compiler doesn't always detect when you mix them up, so make sure you are using the correct one.

Trap: `^` is **not** exponentiation! `2^2` is not `4`. As we will see later, it is `0`.

The Ternary Operator

The ternary operator `?:` is a compact way to choose between two values based on a condition.

```
condition ? value_if_true : value_if_false
```

```
int temperature = 30;
std::string weather = (temperature > 25) ? "hot" : "cool";
std::cout << "it is " << weather << std::endl;
```

This is equivalent to an if-else, but in a single expression.

```
// same thing with if-else
std::string weather;
if (temperature > 25) {
    weather = "hot";
} else {
    weather = "cool";
}
```

The ternary operator is best for simple choices. If the logic is complex, use a regular if-else — readability matters more than brevity.

Operator Precedence

When an expression has multiple operators, C++ uses precedence rules to determine the order of evaluation. Operators with higher precedence are evaluated first.

Here is a simplified precedence table, from highest to lowest.

Precedence	Operators	Description
1	`() [] . ->`	grouping, subscript
2	`++ --` (postfix)	postfix increment
3	`++ --` (prefix) `! ~ -` | prefix, unary |	
4	`* / %`	multiplicative
5	`+ -`	additive
6	`<< >>`	shift
7	`< <= > >=` | relational |	
8	`== !=`	equality
9	`&`	bitwise AND
10	`^`	bitwise XOR
11	`|`	bitwise OR
12	`&&`	logical AND
13	`||`	logical OR
14	`?:`	ternary
15	`= += -= *=` etc. | assignment |	

The classic gotcha is forgetting that comparison binds tighter than bitwise operators.

```
// BUG: this checks (flags) & (2 == 2), not (flags & 2) == 2
if (flags & 2 == 2) { ... }

// CORRECT:
if ((flags & 2) == 2) { ... }
```

Tip: When in doubt, use parentheses. They make your intent clear to both the compiler and anyone reading your code. You do not get bonus points for memorizing the precedence table.

Try It

Here is a small program that uses several operators. Try modifying it and predicting the output before running it.

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

int main()
{
    int x = 10;
    int y = 3;

    std::cout << "x + y  = " << x + y << std::endl;
    std::cout << "x / y  = " << x / y << std::endl;
    std::cout << "x % y  = " << x % y << std::endl;

    x += 5;
    std::cout << "x += 5 => " << x << std::endl;

    int a = 5;
    int b = ++a;
    int c = a++;
    std::cout << "a=" << a << " b=" << b
              << " c=" << c << std::endl;

    std::string result = (x > y) ? "Jump around!" : "U can't touch this";
    std::cout << result << std::endl;

    return 0;
}
```

Key Points

- `=` is assignment; `==` is comparison. Confusing them is one of the most common bugs.
- Integer division (`/` with two integers) truncates the result. Use a floating-point operand if you need the decimal part.
- `%` (modulo) gives the remainder of integer division. It only works with integers.
- Logical operators `&&` and `||` short-circuit: they stop evaluating as soon as the result is determined.
- Prefix `++n` increments then returns the new value; postfix `n++` returns the old value then increments.
- Compound assignment operators like `+=` are shorthand for `x = x + value`.
- Bitwise operators work on individual bits and are essential for low-level programming.
- The ternary operator `?:` is a concise alternative to simple if-else statements.
- When operator precedence is not obvious, use parentheses to make your intent clear.

Exercises

1. What is the difference between `7 / 2` and `7.0 / 2` in C++? Why does it matter?
2. What does the following code print?

   ```cpp
   int a = 10;
   int b = a++;
   int c = ++a;
   std::cout << a << " " << b << " " << c << std::endl;
   ```

3. What is the value of each expression?
   - `17 % 5`
   - `20 % 4`
   - `3 % 7`
4. What does this expression evaluate to?

   ```
   int x = 0;
   bool result = (x != 0) && (100 / x > 5);
   ```

   Why does it not crash even though `x` is `0`?
5. Where is the bug?

   ```
   int x = 5;
   if (x = 10) {
       std::cout << "x is 10" << std::endl;
   }
   ```
6. Where is the bug?

   ```
   int flags = 10;
   if (flags & 2 == 2) {
       std::cout << "bit is set" << std::endl;
   }
   ```
7. What does this code print?

   ```
   int score = 85;
   std::string grade = (score >= 90) ? "A"
                     : (score >= 80) ? "B"
                     : (score >= 70) ? "C"
                     : "F";
   std::cout << grade << std::endl;
   ```
8. Write a short program that asks the user for an integer and prints whether it is even or odd, positive or negative (or zero), using the modulo and comparison operators.
9. **What does this print?**

   ```
   #include <iostream>

   int main()
   {
       int x = 10;
       x += 5;
       x *= 2;
       x -= 3;
       x /= 4;
       x %= 5;
       std::cout << x << "\n";
       return 0;
   }
   ```

   Walk through each line and show the value of `x` after that line runs.
10. **Think about it:** Without using parentheses, what does C++ make of the following expression?

    ```
    bool result = a < b && c == d || !e;
    ```

List the operators in the order C++ evaluates them, then rewrite the expression with parentheses that make the precedence explicit. Why is the second form preferable even though both produce the same result?