Numbers in C++

February 28, 2026

Numbers

There are only 10 kinds of people in the world: those who understand binary and those who don't.

If that joke doesn't make sense yet, it will by the end of this chapter.

At its heart, a CPU is a super-fast glorified calculator. Everything it does — drawing pixels on your screen, playing your favorite Depeche Mode track, sending a message — ultimately boils down to operations on numbers. When we think about the number five, we might write `5`, or `V`, or `|||||`, but they all represent the same thing. You already saw in Chapter 9 that the character `'5'` is actually stored as the ASCII code `53`. There are many ways to represent a number, and in this chapter you will learn why that matters and how C++ lets you work with them.

1. Bases

Let's extend that joke. *There are only 10 kinds of people in the world: those who understand binary, those who don't, and those who weren't expecting this joke to be in base 3. And those who weren't expecting base 4. And those who...*

You get the idea. The punchline changes depending on which **base** (or **radix**) you are using.

Decimal (Base 10)

You have been using decimal your whole life. It uses ten digits: `0` through `9`. Each position represents a power of 10:

```
4   7   2
|   |   |
|   |   +-- 2 × 10^0 =   2
|   +------ 7 × 10^1 =  70
+---------- 4 × 10^2 = 400
                       ---
```

```
472
```

A number is divisible by 10 when its last digit is `0`. You can tell if a number is divisible by 2 by checking whether its last digit is even. Divisibility by 8 is trickier — you have to check the last three digits. These rules come directly from how place values work.

Binary (Base 2)

It is often said that computers think in 1s and 0s. That is not entirely accurate, but at the lowest level, data is stored in **bits** — each one either `0` or `1`. Binary is base 2, so each position represents a power of 2:

```
  1   0   1   0   1   0
  |   |   |   |   |   |
  |   |   |   |   |   +-- 0 × 2^0 =  0
  |   |   |   |   +------ 1 × 2^1 =  2
  |   |   |   +---------- 0 × 2^2 =  0
  |   |   +-------------- 1 × 2^3 =  8
  |   +------------------ 0 × 2^4 =  0
  +---------------------- 1 × 2^5 = 32
                                     --
                                     42
```

Counting in binary looks like this:

```
Decimal:  0   1   2   3   4   5   6   7   8
Binary:   0   1  10  11 100 101 110 111 1000
```

In binary, you can tell if a number is divisible by 2 by checking the last bit — if it is `0`, the number is even. Divisible by 8? Check the last three bits. But divisibility by 10 is no longer obvious at a glance.

> **Tip:** Notice the symmetry. In decimal, divisibility by the base (10) is trivial to check. In binary, divisibility by the base (2) is trivial. Each system makes certain things easy and others hard.

Hexadecimal (Base 16)

Binary numbers get long quickly. The number `255` is `11111111` in binary — eight digits for what decimal handles in three. **Hexadecimal** (hex) uses sixteen digits: `0`–`9` and `A`–`F` (where `A` = 10, `B` = 11, ... `F` = 15). Each hex digit represents exactly four bits:

```
Binary:  1010 1100
Hex:        A    C  ->  0xAC = 172
```

This makes hex a compact way to write binary values. Two hex digits represent one byte (8 bits), and eight hex digits represent a 32-bit integer. You will see

hex used frequently for colors, memory addresses, and bit masks.

Octal (Base 8)

Octal uses eight digits: `0`–`7`. Each octal digit represents exactly three bits:

```
Binary:  101 010
Octal:     5   2  ->  052 = 42
```

Octal is less common than hex in modern code, but you will encounter it when working with Unix file permissions (like `0755`).

Try It: Counting in Binary

This program counts from 0 to 15 and prints each number in decimal, binary, hex, and octal so you can see the patterns side by side:

```cpp
#include <print>

int main() {
    std::println("{:>4} {:>8} {:>4} {:>4}", "Dec", "Binary", "Hex", "Oct");
    std::println("{:->4} {:->8} {:->4} {:->4}", "", "", "", "");
    for (int i = 0; i <= 15; ++i) {
        std::println("{:4d} {:08b} {:4x} {:4o}", i, i, i, i);
    }
}
```

2. Literals in Other Bases

C++ lets you write integer literals in binary, hexadecimal, and octal using prefixes. You first used integer literals in Chapter 2 when you initialized variables like `int x = 42;` — that `42` is a decimal literal.

```cpp
int dec = 42;         // decimal (no prefix)
int bin = 0b101010;   // binary  (0b prefix)
int hex = 0x2A;       // hex     (0x prefix)
int oct = 052;        // octal   (0  prefix)
```

All four variables hold exactly the same value: `42`. The prefix only affects how you write the number in your source code, not how it is stored.

> **Tip:** Be careful with leading zeros. `052` is **not** decimal `52` — it is octal `52`, which equals decimal `42`. This is a common source of confusion. If you want decimal 52, write `52` without a leading zero.

Digit Separators

Large numbers can be hard to read. C++14 introduced the single-quote `'` as a digit separator, which you can place anywhere between digits for readability:

```
int billion    = 1'000'000'000;    // easier to read than 1000000000
int bits       = 0b1010'1100;      // group binary by nibbles (4 bits)
int color      = 0xFF'80'00;       // group hex by byte
```

The separator has no effect on the value — the compiler ignores it completely.

Try It: Same Value, Different Spellings

This program shows that no matter how you write a literal, the compiler stores the same number:

```
#include <print>

int main() {
    int dec = 1984;
    int bin = 0b11111000000;
    int hex = 0x7C0;
    int oct = 03700;

    std::println("The year George Orwell warned us about:");
    std::println("  Decimal:  {}", dec);
    std::println("  Binary:   {}", bin);
    std::println("  Hex:      {}", hex);
    std::println("  Octal:    {}", oct);
    std::println("  All equal? {}", dec == bin && bin == hex && hex == oct);
}
```

3. Printing in Other Bases

C++23 gives you `std::format` and `std::println` with format specifiers that make it easy to print numbers in different bases:

```
int val = 42;
std::println("Decimal:     {}", val);      // 42
std::println("Binary:      {:b}", val);    // 101010
std::println("Hex:         {:x}", val);    // 2a
std::println("Hex (upper): {:X}", val);    // 2A
std::println("Octal:       {:o}", val);    // 52
```

You can also add a `#` flag to include the base prefix in the output:

```
std::println("Binary:  {:#b}", val);   // 0b101010
std::println("Hex:     {:#x}", val);   // 0x2a
std::println("Octal:   {:#o}", val);   // 052
```

Tip: The format specifiers `{:b}`, `{:x}`, and `{:o}` display the value in a different base, but they do not change the value itself. The variable still holds the same number — you are just looking at it from a different angle.

Try It: Number Viewer

This program asks for a number and displays it in every base:

```
#include <iostream>
#include <print>

int main() {
    std::print("Enter a number: ");
    int val{};
    std::cin >> val;

    std::println("Decimal:  {}", val);
    std::println("Binary:   {:#b}", val);
    std::println("Hex:      {:#x}", val);
    std::println("Octal:    {:#o}", val);
}
```

4. Converting from Other Bases

Manual Conversion

To convert a number from another base to decimal, multiply each digit by its place value and add the results. You already saw this with binary:

```
0x2A = 2 × 16 + 10 × 1 = 42
052  = 5 × 8  +  2 × 1  = 42
```

To convert from decimal to another base, repeatedly divide by the base and collect the remainders:

```
42 / 2 = 21 remainder 0
21 / 2 = 10 remainder 1
10 / 2 =  5 remainder 0
 5 / 2 =  2 remainder 1
 2 / 2 =  1 remainder 0
 1 / 2 =  0 remainder 1
Read remainders bottom-to-top: 101010
```

`std::stoi` with a Base

The `std::stoi` function (string-to-integer) accepts an optional base parameter. You first used `std::stoi` to convert strings to numbers — now you can tell it which base the string is in:

```
int a = std::stoi("101010", nullptr, 2);   // binary  -> 42
int b = std::stoi("2A", nullptr, 16);      // hex     -> 42
int c = std::stoi("52", nullptr, 8);       // octal   -> 42
```

The second parameter is a pointer that receives the position where parsing stopped — `nullptr` means you don't need it. The third parameter is the base.

> **Tip:** `std::stoi` throws `std::invalid_argument` if the string cannot be parsed and `std::out_of_range` if the value is too large. Always be prepared to handle these when converting user input.

Try It: Base Converter

This program converts strings in different bases to decimal:

```
#include <print>
#include <string>

int main() {
    // "Take On Me" was released in 1985
    std::string hex_str = "7C1";     // 1985 in hex
    int val = std::stoi(hex_str, nullptr, 16);
    std::println("Hex \"{}\" = decimal {}", hex_str, val);

    std::string bin_str = "11111000001";  // 1985 in binary
    val = std::stoi(bin_str, nullptr, 2);
    std::println("Binary \"{}\" = decimal {}", bin_str, val);

    std::string oct_str = "3701";    // 1985 in octal
    val = std::stoi(oct_str, nullptr, 8);
    std::println("Octal \"{}\" = decimal {}", oct_str, val);
}
```

5. Two's Complement

So far we have only talked about positive numbers. But how does the computer represent negative numbers? You can't just put a minus sign in front of a binary number — there are only 1s and 0s.

One's Complement (and Why We Don't Use It)

An early idea was **one's complement**: flip every bit to get the negative. In an 8-bit system:

```
 42 = 0010 1010
-42 = 1101 0101  (every bit flipped)
```

This mostly works, but it has a fatal flaw: there are two representations of zero.

```
+0 = 0000 0000
-0 = 1111 1111
```

Having two zeros complicates hardware and arithmetic. Is `-0 == +0`? It should be, but the bit patterns differ. Engineers needed a better solution.

Two's Complement

Two's complement fixes this by adding one extra step: flip all the bits *and* add 1.

```
 42 = 0010 1010
      1101 0101  (flip bits)
    + 0000 0001  (add 1)
      ---------
-42 = 1101 0110
```

Now there is only one zero:

```
 0 = 0000 0000
     1111 1111  (flip bits)
   + 0000 0001  (add 1)
     ---------
     0000 0000  (overflow discarded - still 0!)
```

In two's complement, the highest bit (the **sign bit**) tells you whether the number is negative. For an 8-bit signed integer:

- `0xxx xxxx` — positive (0 to 127)
- `1xxx xxxx` — negative (-128 to -1)

This gives 8-bit signed integers a range of **-128 to 127**. Notice that there is one more negative value than positive — that's because zero takes one of the "positive" slots.

> **Tip:** Nearly every modern computer uses two's complement for signed integers. When you declare `int x = -42;`, the bit pattern stored in memory is the two's complement representation.

Why Two's Complement Is Brilliant

The beauty of two's complement is that addition and subtraction **just work** with the same hardware used for unsigned numbers. The CPU does not need separate circuitry for signed math — it uses the same adder for both.

Try It: Seeing Two's Complement

This program shows the bit patterns of positive and negative numbers so you can see two's complement in action:

```
#include <cstdint>
#include <print>
```

```cpp
int main() {
    int8_t values[] = {42, -42, 0, -1, 127, -128};
    for (int8_t i : values) {
        // cast to unsigned to see the raw bit pattern
        uint8_t bits = static_cast<uint8_t>(i);
        std::println("{:4d} = {:08b}", i, bits);
    }
}
```

Output:

```
  42 = 00101010
 -42 = 11010110
   0 = 00000000
  -1 = 11111111
 127 = 01111111
-128 = 10000000
```

6. Integer Sizes and Ranges

You saw that an 8-bit signed integer can hold values from -128 to 127. But `int` is not 8 bits — so how big is it, and what range can it hold?

Bytes

A group of 8 bits is called a **byte**, and it is the basic unit of measurement for computer memory. When you buy a computer with 16 GB of RAM, you are buying roughly 16 billion bytes of memory. More precisely, 1 GB (gigabyte) is 2^{30} bytes, which equals 1,073,741,824 — just over a billion. So 16 GB is 17,179,869,184 bytes. Close enough to 16 billion for casual conversation, but the difference matters when you are counting precisely.

You will also encounter kilobytes (KB, 2^{10} = 1,024 bytes), megabytes (MB, 2^{20} = 1,048,576 bytes), and terabytes (TB, 2^{40}). Notice a pattern: each unit is a power of 2, not a power of 10, because binary makes powers of two the natural grouping.

> **Tip:** The `sizeof` operator in C++ returns sizes in bytes, not bits. Since 1 byte = 8 bits, a 4-byte `int` has 32 bits. You will see `sizeof` used frequently when working with memory and data structures.

How Many Bits?

C++ has several integer types, each with a minimum guaranteed size. On most modern systems, the sizes are:

Type	Typical Size	Bits
`char`	1 byte	8
`short`	2 bytes	16
`int`	4 bytes	32
`long`	4 or 8 bytes	32 or 64
`long long`	8 bytes	64

You can check the size of any type with the `sizeof` operator:

```
std::println("char:      {} bytes", sizeof(char));       // 1
std::println("short:     {} bytes", sizeof(short));      // 2
std::println("int:       {} bytes", sizeof(int));        // 4
std::println("long long: {} bytes", sizeof(long long)); // 8
```

The Range Formula

With `n` bits, you can represent 2^n distinct values. How those values are divided depends on whether the type is signed or unsigned:

- **Unsigned** (no negatives): 0 to 2^n - 1
- **Signed** (two's complement): $-2^{(n-1)}$ to $2^{(n-1)}$ - 1

For example, with 8 bits:

- Unsigned: 0 to 255 (2^8 - 1)
- Signed: -128 to 127 (-2^7 to 2^7 - 1)

Here are the ranges for the common types:

Type	Range
`unsigned char`	0 to 255
`signed char`	-128 to 127
`unsigned short`	0 to 65,535
`short`	-32,768 to 32,767
`unsigned int`	0 to 4,294,967,295 (about 4.3 billion)
`int`	-2,147,483,648 to 2,147,483,647 (about +/- 2.1 billion)
`unsigned long long`	0 to 18,446,744,073,709,551,615
`long long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Tip: If you are ever unsure about a type's range, use `std::numeric_limits` from `<limits>`:

```
std::println("int max: {}", std::numeric_limits<int>::max());
std::println("int min: {}", std::numeric_limits<int>::min());
```

Signed vs. Unsigned

Every integer type has a signed and unsigned variant. You first saw `unsigned` in Chapter 4 with `size_t`, the type returned by `.size()` on containers.

```
unsigned int positive_only = 42;
int can_be_negative = -42;
```

An unsigned type gives up negative values in exchange for a larger positive range. A 32-bit `unsigned int` can hold values up to about 4.3 billion, while a signed `int` tops out around 2.1 billion.

> **Tip:** Mixing signed and unsigned types in comparisons can cause surprising bugs. The expression `-1 < 0u` is `false` because `-1` is implicitly converted to a very large unsigned value. When possible, stick to signed types for general arithmetic and use unsigned only when you have a specific reason (like bit manipulation or interfacing with APIs that require it).

What Happens When You Overflow?

If you try to store a value that does not fit, the behavior depends on whether the type is signed or unsigned:

- **Unsigned overflow** wraps around. Adding 1 to the maximum value gives 0:

```
unsigned char x = 255;
x = x + 1;  // x is now 0 (wraps around)
```

- **Signed overflow** is undefined behavior. The compiler can do anything:

```
int y = 2'147'483'647;  // INT_MAX
y = y + 1;               // undefined behavior!
```

> **Tip:** "Undefined behavior" is not just a theoretical concern. Compilers actively exploit it for optimization. A signed overflow might silently wrap, or it might cause your program to behave in completely unexpected ways. Never rely on signed overflow.

Try It: Exploring Sizes and Limits

This program prints the size and range of each integer type:

```
#include <limits>
#include <print>

int main() {
    std::println("{:<12} {:>5} {:>22} {:>22}",
                 "Type", "Bytes", "Min", "Max");
```

```
    std::println("{:<12} {:>5} {:>22} {:>22}", "char",
        sizeof(char),
        std::numeric_limits<signed char>::min(),
        std::numeric_limits<signed char>::max());

    std::println("{:<12} {:>5} {:>22} {:>22}", "short",
        sizeof(short),
        std::numeric_limits<short>::min(),
        std::numeric_limits<short>::max());

    std::println("{:<12} {:>5} {:>22} {:>22}", "int",
        sizeof(int),
        std::numeric_limits<int>::min(),
        std::numeric_limits<int>::max());

    std::println("{:<12} {:>5} {:>22} {:>22}", "long long",
        sizeof(long long),
        std::numeric_limits<long long>::min(),
        std::numeric_limits<long long>::max());
}
```

7. Binary Addition and Subtraction

Binary Addition

Binary addition works just like decimal addition, but you carry at 2 instead of 10:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10  (0, carry 1)
```

Let's add `42 + 15` in 8-bit binary:

```
  0010 1010   (42)
+ 0000 1111   (15)
-----------
  0011 1001   (57)
```

Subtraction with Two's Complement

To subtract, you negate the second number (using two's complement) and add. Let's compute `42 - 15`:

Step 1 — find `-15` using two's complement:

```
15 = 0000 1111
     1111 0000   (flip bits)
```

```
     + 0000 0001   (add 1)
       ---------
 -15 = 1111 0001
```

Step 2 — add `42 + (-15)`:

```
    0010 1010   (42)
  + 1111 0001   (-15)
  -----------
  1 0001 1011
  ^
  overflow bit (discarded in 8 bits)
```

The result is `0001 1011` = 27. Correcto! The overflow bit is discarded because we are working with 8-bit values, and the addition produces the right answer automatically.

Let's also try a negative result: `15 - 42`:

Step 1 — find `-42`:

```
  42 = 0010 1010
       1101 0101   (flip bits)
     + 0000 0001   (add 1)
       ---------
 -42 = 1101 0110
```

Step 2 — add `15 + (-42)`:

```
    0000 1111   (15)
  + 1101 0110   (-42)
  -----------
    1110 0101   (-27)
```

The sign bit is `1`, so the result is negative. To verify, convert back: flip the bits (`0001 1010`), add 1 (`0001 1011` = 27), so the answer is `-27`. Perfecto.

> **Tip:** Overflow can happen when the result is too large (or too small) for the number of bits. Adding two large positive numbers can wrap around to a negative value. In C++, signed integer overflow is **undefined behavior** — the compiler is free to do anything. Be careful with arithmetic near the limits of a type.

Try It: Unsigned Wraparound

This program demonstrates unsigned addition wrapping around, the same way the CPU adds binary numbers:

```cpp
#include <cstdint>
#include <print>

int main() {
```

```
    uint8_t a = 200;
    uint8_t b = 100;
    uint8_t result = a + b;  // 300 doesn't fit in 8 bits

    std::println("{} + {} = {} (wrapped)", a, b, result);
    // 200 + 100 = 44 (wrapped), because 300 - 256 = 44

    uint8_t x = 42;
    uint8_t y = 15;
    std::println("{} + {} = {}", x, y, static_cast<int>(x + y));
    std::println("{} - {} = {}", x, y, static_cast<int>(x - y));
}
```

8. Shift Operators

The shift operators `<<` and `>>` move bits left or right by a specified number of positions. You first saw `<<` and `>>` used for stream I/O — here we are talking about their original purpose as bitwise operators. For a deeper look at all the bitwise operators, see the Operators chapter.

Left Shift: `<<`

Left shift moves every bit to the left and fills the empty positions on the right with zeros:

```
  0000 0101  (5)
  << 1
  0000 1010  (10)

  0000 0101  (5)
  << 3
  0010 1000  (40)
```

Each left shift by 1 **multiplies the value by 2**. Shifting left by `n` is the same as multiplying by 2^n:

```
int x = 5;
int doubled  = x << 1;   // 10  (5 × 2)
int times8   = x << 3;   // 40  (5 × 2 × 2 × 2)
```

Right Shift: `>>`

Right shift moves every bit to the right. For unsigned values, zeros fill in from the left. For signed values, the sign bit is copied (called **arithmetic shift**), so negative numbers stay negative after a right shift.

```
  0010 1000  (40)
  >> 1
```

```
  0001 0100  (20)

  0010 1000  (40)
  >> 3
  0000 0101  (5)
```

Each right shift by 1 **divides the value by 2** (discarding any remainder). Shifting right by `n` is the same as dividing by 2^n:

```
int y = 40;
int halved    = y >> 1;   // 20  (40 / 2)
int div8      = y >> 3;   // 5   (40 / 8)
```

What About Odd Numbers?

When you right-shift an odd number, the lowest bit is lost — just like integer division discards the remainder:

```
int odd = 7;
int result = odd >> 1;  // 3, not 3.5 (same as 7 / 2)
```

Compound Assignment

As with other operators, there are compound assignment forms:

```
int flags = 1;
flags <<= 4;   // flags is now 16  (1 shifted left 4)
flags >>= 2;   // flags is now 4   (16 shifted right 2)
```

Tip: Shifting by a negative amount or by more bits than the type has is **undefined behavior**. For a 32-bit `int`, valid shift amounts are 0 to 31.

Tip: Modern compilers optimize multiplication and division by powers of two into shift operations automatically. Write `x * 4` rather than `x << 2` unless you are doing actual bit manipulation — it is clearer, and the compiler will generate the same code.

Try It: Powers of Two with Shifts

This program uses shifts to compute and display powers of two:

```
#include <print>

int main() {
    std::println("Powers of two using left shift:");
    for (int i = 0; i < 16; ++i) {
        int power = 1 << i;
        std::println("  1 << {:2} = {:5}  ({:#018b})", i, power, power);
    }
```

```
    std::println("\nDividing 1000 by powers of two using right shift:");
    int val = 1000;
    for (int i = 0; i <= 4; ++i) {
        std::println("  {} >> {} = {}", val, i, val >> i);
    }
}
```

9. Conclusion

Here are the key takeaways from this chapter:

- **A number is a number**, regardless of how you represent it. `42`, `0b101010`, `0x2A`, and `052` are all the same value.
- **Decimal, binary, hex, and octal** are just different bases. Each one makes certain patterns easier to see.
- **C++ supports multiple bases** in literals (`0b`, `0x`, `0`), output (`{:b}`, `{:x}`, `{:o}`), and conversion (`std::stoi` with a base).
- **A byte is 8 bits**, and integer types come in different sizes — from 1-byte `char` to 8-byte `long long`. The number of bits determines the range of values a type can hold.
- **Two's complement** is how computers represent negative integers. It eliminates the double-zero problem and lets addition and subtraction share the same hardware.
- **Signed overflow is undefined behavior**, but unsigned overflow wraps around predictably. Be careful with arithmetic near the limits of a type.
- **Shift operators** (`<<`, `>>`) move bits and are equivalent to multiplying and dividing by powers of two.

A number can wear many different outfits, but underneath, es el mismo numero. No te preocupes — you have got this. Nos vemos en el proximo capitulo!

Content outline and editorial support from Ben. Words by Claude, the Opus.