Gorgo Starting C++

April 11, 2026

Contents

Author Intro

C++ is probably the most complicated, powerful, and confusing language you will ever learn. It is the language of choice for developers looking for power and efficiency in embedded systems and well as large scale backend systems. Very few people understand everything, so as a beginner seek to understand and be comfortable with enough to do the things you want to do. Learning C++ also has the advantage of introducing you to a wide variety of programming concepts.

this booklet is to be experimented with. if you want a book to read, these are not the droids you are looking for. if you want something to introduce you to key concepts of C++ with some pointers for experiments you can try yourself, read on and write some code! programming is learned by writing code; there is just no other way that comes close to it. not everyone starts out as a great programmer, but everyone can become one by writing code either from scratch or by modifying code of others.

i have avoided making code example easily copy and pasteable since the exercise of manually copying copy and typing it in will help you get accustom to syntax and patterns.

i hope this booklet will be something that can help you learn to love the challenge/satisfaction/power/rewards that come with being a great programmer.

ben

0. How to use this booklet

This is a short booklet to help a new programmer learn C++.

Each chapter is meant to help you understand a topic, but you will still want to reference API descriptions for more specifics of the parameters and operating conditions. Hopefully, you'll have the context you need to understand API documentation.

Tips

Tips call out details that you need to pay special attention to. **Traps** warn you of common mistakes made. **Wut** calls out a detail that is counter-intuitive, so make sure you pay attention.

Function Signatures

When this book introduces a new function, it shows the function's **signature** and return type. A function's signature is its name and parameter list — it uniquely identifies the function. We also show the return type so you know what the function gives back. For example:

```
int abs(int n);
```

This tells you that `abs` takes one `int` parameter and returns an `int`. You do not need to understand every detail the first time you see it, but it gives you three things at a glance: what the function is called, what goes in, and what comes out. As you work through API documentation on your own, signatures are the first thing you will look at, so getting comfortable reading them early pays off.

Try It

As the intro to the most amazing programming language book ever written [1] starts out:

> The only way to learn a new programming language is by writing programs in it.

You need to write some code. Make sure you try writing some programs from scratch. At the end of most sections is a starter program that you can type in and modify to play with. Don't use it as an excuse to avoid writing some of your own starter programs. It's the only way to master a language.

Exercises

Don't skip the exercises at the end of the chapters. You can get the answer key, but don't look at the answer key before you work out the answer yourself. If you look at the answer key first, the concepts will not sink in.

1. Introduction

Welcome to C++. A program that cannot communicate is a black box — it runs, but you have no way to see what it did or tell it what to do. Input and output are the first skills every programmer needs. In this chapter you will write your first program, learn how to compile and run it, and get comfortable with basic I/O using `std::cout` and `std::cin`. By the end, you will be able to write programs that talk to the user and respond to command-line arguments.

Hello, World!

Every programming journey begins with the same tradition: printing "Hello, World!" to the screen. Here is what that looks like in C++:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

That is not a lot of code, but there is a lot going on. Let's break it down.

`#include <iostream>` tells the compiler to include the **iostream** library, which gives you tools for input and output. Think of it as importing functionality that someone else already wrote for you.

`int main()` is the entry point of every C++ program. When you run your program, execution starts here. The `int` before `main` means the function returns an integer value. By convention, returning `0` means the program finished successfully.

`std::cout << "Hello, World!" << std::endl;` is the line that actually prints text to the screen. We will explain what `std::` means shortly. The `<<` operator sends data to the output stream.

`std::cout << "Hello, World!"` writes `Hello, World!` to the screen, with two caveats. First, `Hello World!` is not on its own line. If something else is output, it will appear on the same line right after `!`. Second, `cout` uses **buffer**ing — a technique to improve performance by collecting data to output, so that it can do fewer writes to the terminal. When writing to an actual terminal, starting a new line will also cause **buffered** – collected – data to be written to the screen. `<< std::endl` addresses both of these issues. It starts a new line so that subsequent text will be written to the next line of output, and it **flush**es the buffer by immediately writing any previously buffered output.

The `<<` operator has this signature – this may not mean much to you now, but for consistency we are showing you here:

```cpp
std::ostream& operator<<(std::ostream& os, const T& value);
```

It takes an output stream and a value, writes the value to the stream, and returns the stream so you can chain multiple `<<` together.

`std::endl` ends the line and flushes the output buffer. It is actually a function with this signature:

```cpp
std::ostream& endl(std::ostream& os);
```

It writes a newline character and then flushes the stream.

Tip: You can also use `"\n"` instead of `std::endl` to end a line. The difference is that `std::endl` flushes the output buffer, while `"\n"` does not. For most programs, `"\n"` is fine and slightly faster.

Semicolons and Curly Braces

Every statement in C++ ends with a semicolon (`;`). A statement is a single instruction — printing text, declaring a variable, returning a value. Forgetting a semicolon is one of the most common mistakes beginners make, and the compiler error message can be confusing because it often points to the *next* line rather than the line with the missing semicolon.

Curly braces (`{` and `}`) define a **block** of code. In the Hello World program, the braces after `int main()` mark the beginning and end of the `main` function's body. Everything between `{` and `}` belongs to that function.

```
int main()
{                   // start of block
    // statements go here
    return 0;
}                   // end of block
```

You will see curly braces everywhere in C++ — around functions, loops, and if statements. They always come in pairs: every `{` needs a matching `}`.

Tip: Indent the code inside curly braces to make it easier to read. Most C++ programmers use 4 spaces per level of indentation.

Compiling and Running

C++ is a **compiled** language. You write your source code in a text file, then use a compiler to translate it into a program your computer can run.

Save the Hello World program to a file called `hello.cpp`. C++ source files typically end in `.cpp`.

To compile it, open a terminal and run:

```
c++ -o hello hello.cpp
```

This tells the compiler to take `hello.cpp` and produce an executable called `hello`. The `-o hello` part names the output file.

Now run it:

```
./hello
Hello, World!
```

Congratulations, you just compiled and ran your first C++ program!

Tip: Always compile with warnings enabled. Use `c++ -Wall -Wextra -pedantic -o hello hello.cpp` to catch potential problems early. The compiler is your friend — listen to its warnings.

If you get an error, read the error message carefully. The compiler will tell you the file name and line number where the problem is. Common mistakes include forgetting a semicolon at the end of a line or misspelling `iostream`.

Namespaces

You have been typing `std::cout` and `std::endl`, and you might be wondering what the `std::` part is about.

C++ uses **namespaces** to organize code and avoid naming conflicts. The standard library puts all of its names inside a namespace called `std`. When you write `std::cout`, you are saying "I want the `cout` that lives inside the `std` namespace."

The `::` is called the **scope resolution operator**. You will see it a lot in C++.

If you get tired of typing `std::` everywhere, you can add a **using directive** at the top of your file:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

Now you can write `cout` instead of `std::cout`.

Trap: Using `using namespace std;` in large programs or header files can cause naming conflicts. It is fine for small programs while you are learning, but be aware that many professional C++ programmers avoid it.

For the rest of this book, we will use `std::` explicitly so you always know where names come from.

Output with std::cout

`std::cout` is the **standard output stream**. You send data to it using the `<<` operator, and that data appears on the screen.

You can chain multiple `<<` operators together to print several things on one line:

```
#include <iostream>

int main()
{
    std::cout << "Come as you are" << ", " << "as you were" << std::endl;
    return 0;
}
```

Output:

```
Come as you are, as you were
```

You can also print numbers directly:

```
std::cout << "The year is " << 1991 << std::endl;
```

Output:

```
The year is 1991
```

Escape Sequences

You have already seen `std::endl` end a line. There is another way to do the same thing: put `\n` *inside* the string.

```
std::cout << "Smells Like\nTeen Spirit\n";
```

Output:

```
Smells Like
Teen Spirit
```

The `\n` is not two characters — it is a single character called **newline**. The backslash tells the compiler "the next character is special, do not take it literally." This is called an **escape sequence**.

Why do you need escape sequences at all? Because some characters cannot be written literally inside a string. A double quote (`"`) marks the *end* of a string, so this does not work:

```
std::cout << "She said "hello" and walked away" << std::endl;  // ERROR
```

The compiler sees `"She said "`, thinks the string ends there, and gets confused by the leftover `hello" and walked away"`. You have to **escape** the inner quotes with `\"` to tell the compiler they are part of the string:

```
std::cout << "She said \"hello\" and walked away" << std::endl;
```

Output:

```
She said "hello" and walked away
```

The same problem happens with single quotes inside a **character literal**. A character literal is one character wrapped in single quotes, like `'A'` or `'?'`. Writing `'''` is ambiguous — is the second `'` ending the literal, or is it the character? You escape it with `\'`:

```
char apostrophe = '\'';   // a single-quote character
// no escape needed in a char literal
char quote      = '"';
```

Inside a string literal, the rule flips: `"` needs escaping but `'` does not. Inside a char literal, `'` needs escaping but `"` does not. You only have to escape the delimiter that would otherwise end the literal.

And of course, since `\` itself is the escape character, you need `\\` to write a literal backslash:

```
std::cout << "C:\\Users\\Kurt" << std::endl;
```

Output:

```
C:\Users\Kurt
```

Here is the full list of escape sequences you will see in C++:

Escape	Meaning
`\n`	Newline
`\t`	Horizontal tab
`\r`	Carriage return
`\\`	Backslash
`\"`	Double quote
`\'`	Single quote
`\0`	Null character
`\a`	Alert (bell)
`\b`	Backspace
`\f`	Form feed
`\v`	Vertical tab
`\?`	Question mark

Tip: You will use `\n`, `\t`, `\\`, and `\"` constantly. The others are rare in modern code — `\a` rings the terminal bell, `\b` and `\f` come from the days of teletype printers, and `\?` exists only to defeat an obscure C feature called **trigraphs** that you will probably never see.

Input with std::cin

`std::cin` is the **standard input stream**. It reads data from the keyboard using the `>>` operator, whose signature follows this pattern:

```
std::istream& operator>>(std::istream& is, T& value);
```

It takes an input stream and a variable, reads a value from the stream into the variable, and returns the stream.

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

int main()
{
    std::string name;

    std::cout << "What is your name? ";
    std::cin >> name;
    std::cout << "Hola, " << name << "!" << std::endl;

    return 0;
}
```

When you run this program, it waits for you to type something and press Enter:

```
What is your name? Nirvana
Hola, Nirvana!
```

`std::cin >> name` reads one word from the keyboard and stores it in the variable `name`. We are using `std::string` here to hold text — we will cover strings in detail in a later chapter. For now, just know that you need `#include <string>` to use them.

Trap: `std::cin >>` reads one word at a time, stopping at whitespace. If you type "Los Del Rio", only "Los" would be stored in `name`. To read an entire line, use `std::getline(std::cin, name)` instead.

You can read numbers too:

```
#include <iostream>

int main()
{
    int year;

    std::cout << "What year? ";
    std::cin >> year;
    std::cout << "Dale a tu cuerpo alegria, " << year << "!" << std::endl;

    return 0;
}
```

```
What year? 1996
Dale a tu cuerpo alegria, 1996!
```

Command-Line Arguments

So far, your programs have asked the user for input interactively. But programs can also receive input when they are launched, through **command-line arguments**.

You have already seen `int main()`. There is another form that accepts arguments:

```
#include <iostream>

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cout << "USAGE: " << argv[0] << " <name>" << std::endl;
        return 1;
    }

    std::cout << "Hello, " << argv[1] << "!" << std::endl;
    return 0;
}
```

`argc` is the **argument count** — the number of command-line arguments, including the program name itself. `argv` is the **argument vector** — an array of strings containing each argument.

- `argv[0]` is always the program name
- `argv[1]` is the first argument the user provides
- `argv[2]` would be the second, and so on

If you compile this as `greet` and run it:

```
./greet Kurt
Hello, Kurt!
```

If you run it without an argument:

```
./greet
USAGE: ./greet <name>
```

The program checks if `argc < 2` — meaning no argument was provided — and prints a **USAGE message** to tell the user how to run the program correctly. Returning `1` instead of `0` signals that something went wrong.

Tip: Always validate command-line arguments before using them. Accessing `argv[1]` when no argument was provided leads to undefined behavior — your program might crash, or worse, silently do the wrong thing.

Try It

Here is a starter program that combines what you have learned. Type it in, compile it, and experiment with it. Try changing the output, adding more `cin` reads, or using command-line arguments.

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

int main()
{
    std::string song;
    int year;

    std::cout << "Name a 90s song: ";
```

```
    std::getline(std::cin, song);

    std::cout << "What year? ";
    std::cin >> year;

    std::cout << song << " (" << year
              << ") es una cancion increible!"
              << std::endl;

    return 0;
}
```

```
Name a 90s song: Smells Like Teen Spirit
What year? 1991
Smells Like Teen Spirit (1991) es una cancion increible!
```

Key Points

- Every C++ program starts at `main()`.
- Every statement ends with a semicolon (`;`).
- Curly braces (`{}`) define blocks of code — they group statements together.
- `#include <iostream>` gives you access to `std::cout` and `std::cin`.
- `std::cout << value` prints to the screen; `std::cin >> variable` reads from the keyboard.
- **Escape sequences** like `\n`, `\t`, `\"`, and `\\` let you put special characters inside string and character literals.
- C++ source files end in `.cpp` and are compiled with `c++`.
- Namespaces like `std::` organize code and prevent naming conflicts.
- Command-line arguments are accessed through `argc` and `argv`.
- Always validate command-line arguments before using them.
- Always compile with warnings enabled.

Exercises

1. What does the following program print?

   ```
   #include <iostream>

   int main()
   {
       std::cout << "A" << "B" << std::endl;
       std::cout << "C" << std::endl;
       return 0;
   }
   ```

2. What is wrong with the following program?

   ```
   #include <iostream>

   int main()
   {
       std::cout << "Here we are now" << std::endl
       return 0;
   }
   ```

3. Why does `std::cout` have `std::` in front of it? What would happen if you removed the `std::` without adding a `using namespace std;` directive?

4. When you compile a program with `c++ -o hello hello.cpp`, what does the `-o hello` part do? What would happen if you left it out?

5. Consider the following program:

```cpp
#include <iostream>

int main(int argc, char *argv[])
{
    std::cout << argv[2] << std::endl;
    return 0;
}
```

   What happens if you run it with `./program alpha beta gamma`? What happens if you run it with `./program alpha`?

6. If `argc` is 4, how many arguments did the user provide on the command line (not counting the program name)?

7. Write a program that asks for the user's name and favorite number, then prints a message using both. For example: "Hola, Carlos! Your favorite number is 7."

8. **Think about it:** What is the difference between writing `std::endl` and writing `"\n"` at the end of a line? When does it actually matter, and when is it the same?

9. **Where is the bug?** A program does this:

```cpp
#include <iostream>

int main()
{
    std::cout << "Loading...";
    // ... pretend a long computation happens here ...
    std::cout << "Done!\n";
    return 0;
}
```

   The user reports that "Loading..." does not appear on the screen until the computation finishes and the program exits. Why does that happen, and how would you fix it so "Loading..." shows up immediately?

10. What does this program print if the user types `Como estas` and presses Enter?

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

int main()
{
    std::string greeting;
    std::cout << "Greeting: ";
    std::cin >> greeting;
    std::cout << "[" << greeting << "]\n";
    return 0;
}
```

    Now change `std::cin >> greeting;` to `std::getline(std::cin, greeting);` and answer the same question. What is the difference, and why?

11. **Where is the bug?** The author wants to print `He said "wassup" and left.` but the compiler refuses to build this:

```cpp
#include <iostream>

int main()
{
    std::cout << "He said "wassup" and left." << std::endl;
    return 0;
}
```

Explain what the compiler sees and rewrite the line so it prints the intended text.

12. What does the following program print?

```cpp
#include <iostream>

int main()
{
    std::cout << "a\\b\tc\nd" << std::endl;
    return 0;
}
```

2. Variables

In Chapter 1 you wrote programs that printed messages and read text from the user. But every value was either a literal typed directly into your source code or a string that was read and immediately used. You had no way to name a value, remember it, or reuse it later. Without that ability, you cannot track a score, accumulate a total, or compare two inputs. Variables solve this — they give a name to a piece of memory so your program can store, retrieve, and update data as it runs. C++ requires each variable to have a **type** so the compiler knows how much memory to allocate and what operations are valid. In this chapter you will learn about the basic types C++ offers, how to declare and use variables, how to group related data with structures, and how to protect values with `const`.

Basic Types

Every variable in C++ has a **type** that determines what kind of data it can hold and how much memory it uses. Here are the fundamental types you will work with:

Integer Types

Integers are whole numbers — no decimal point.

```
int score = 99;
short small_num = 42;
long big_num = 1000000L;
long long very_big = 9000000000LL;
```

The difference between these types is how much memory they use and how large a value they can hold. On most modern systems:

Type	Typical Size	Approximate Range
`short`	2 bytes	-32,768 to 32,767
`int`	4 bytes	-2.1 billion to 2.1 billion
`long`	4 or 8 bytes	at least as large as `int`
`long long`	8 bytes	-9.2 quintillion to 9.2 quintillion

By default, integer types are **signed**, meaning they can hold negative values. If you know a value will never be negative, you can use **unsigned** variants:

```
unsigned int positive_only = 42;
unsigned short flags = 255;
```

An `unsigned int` on a system with 4-byte ints can hold values from 0 to about 4.3 billion, because it does not need to reserve a bit for the sign.

Trap: Be careful mixing signed and unsigned values in comparisons. If you compare a negative `int` with an `unsigned int`, the negative value gets converted to a very large positive number, which is almost certainly not what you want.

Floating-Point Types

Floating-point types store numbers with decimal points.

```
float price = 9.99f;
double pi = 3.14159265358979;
```

`double` gives you more precision than `float` and is the default floating-point type in C++. Unless you have a specific reason to use `float` (like memory constraints), prefer `double`.

Type	Typical Size	Approximate Precision
`float`	4 bytes	~7 decimal digits
`double`	8 bytes	~15 decimal digits

Character Type

A `char` holds a single character and is typically 1 byte.

```
char grade = 'A';
char newline = '\n';
```

Single characters use single quotes. Double quotes are for strings, which are sequences of characters — we will cover those in the next chapter.

Under the hood, a `char` is just a small integer. Every character your computer knows about is assigned a number, and the character is stored in memory as that number. The standard mapping for the basic Latin alphabet, digits, and punctuation is called **ASCII** — the American Standard Code for Information Interchange. ASCII assigns the numbers 0 through 127 to a fixed set of characters: `'A'` is 65, `'a'` is 97, `'0'` is 48, and the space character is 32.

To the CPU, a `char` really is just a number. The CPU has no idea that 65 means the letter A; it sees an integer it can add, subtract, and compare like any other. The interpretation as a letter only happens when the value is sent somewhere that expects characters, like `std::cout`:

```
char letter = 65;
int  number = 65;
std::cout << letter << std::endl;   // prints A
std::cout << number << std::endl;   // prints 65
```

Both variables hold the value 65, but `letter` is a `char` and `number` is an `int`. `std::cout` looks at the type and chooses how to display the value: a `char` becomes a printed glyph, while an `int` becomes digits.

Because a `char` is a number, you can do arithmetic with it:

```
char letter = 'A';
letter = letter + 1;   // letter is now 'B' (66)
```

Adding 1 to `'A'` gives you the next code point, which is `'B'`. Subtracting `'A'` from another letter gives you its position in the alphabet (`'C' - 'A'` is 2). This trick is the basis for many simple text-processing algorithms.

Here is the full ASCII table. The first 32 entries (0 through 31) and entry 127 are **control characters** that do not have a printable glyph; they are listed by their conventional abbreviations (`NUL`, `LF`, `CR`, `ESC`, `DEL`, and so on). Entry 32 (`SP`) is the space character.

```
  0 NUL    16 DLE    32 SP     48 0     64 @     80 P     96 `    112 p
  1 SOH    17 DC1    33 !      49 1     65 A     81 Q     97 a    113 q
  2 STX    18 DC2    34 "      50 2     66 B     82 R     98 b    114 r
  3 ETX    19 DC3    35 #      51 3     67 C     83 S     99 c    115 s
  4 EOT    20 DC4    36 $      52 4     68 D     84 T    100 d    116 t
  5 ENQ    21 NAK    37 %      53 5     69 E     85 U    101 e    117 u
  6 ACK    22 SYN    38 &      54 6     70 F     86 V    102 f    118 v
  7 BEL    23 ETB    39 '      55 7     71 G     87 W    103 g    119 w
  8 BS     24 CAN    40 (      56 8     72 H     88 X    104 h    120 x
  9 HT     25 EM     41 )      57 9     73 I     89 Y    105 i    121 y
 10 LF     26 SUB    42 *      58 :     74 J     90 Z    106 j    122 z
```

```
11 VT     27 ESC    43 +      59 ;      75 K      91 [      107 k     123 {
12 FF     28 FS     44 ,      60 <      76 L      92 \      108 l     124 |
13 CR     29 GS     45 -      61 =      77 M      93 ]      109 m     125 }
14 SO     30 RS     46 .      62 >      78 N      94 ^      110 n     126 ~
15 SI     31 US     47 /      63 ?      79 O      95 _      111 o     127 DEL
```

Tip: ASCII fits in 7 bits, which gives it 128 values. A `char` is 8 bits, leaving room for 128 more values (128 through 255), but those slots are not part of standard ASCII. Different operating systems and locales fill them in different ways, which is one reason text from one system can look like garbage on another. The Numbers chapter has more on how bits, bytes, and ranges work.

Boolean Type

A `bool` holds either `true` or `false`.

```
bool is_alive = true;
bool game_over = false;
```

Booleans are used in conditions and comparisons. We will use them heavily in the Control Flow chapter.

Declaring Variables

When you declare a variable, you are asking the compiler to set aside memory for a value of the specified type and giving that memory a name.

```
int waterfalls = 3;
double speed = 88.0;
char initial = 'T';
```

You can declare multiple variables of the same type on one line:

```
int x = 0, y = 0, z = 0;
```

You can also declare a variable without initializing it:

```
int count;
```

Trap: An uninitialized variable contains whatever garbage was previously in that memory. Using it before assigning a value leads to unpredictable behavior. Always initialize your variables.

In C++, a region of memory with a type is called an **object**. A variable is a name — a label — for an object. When you write `int score = 99`, the compiler allocates memory for an `int` object and gives it the label `score`. This is different from languages like Java or Python, where "object" specifically means an instance of a class. In C++, every variable is a label for an object — `int score = 99` creates an `int` object, `char grade = 'A'` creates a `char` object, and an array of ten `doubles` is an object too. You will hear "object" used this way throughout C++ documentation and error messages, so it is worth knowing early that the word does not imply classes or object-oriented programming.

The `auto` Keyword

When you initialize a variable, the compiler already knows the type of the value on the right-hand side. The `auto` keyword lets you tell the compiler to figure out the type for you:

```
auto waterfalls = 3;        // int (integer literal)
auto speed = 88.0;          // double (floating-point literal)
```

```cpp
auto initial = 'T';         // char (character literal)
auto name = std::string("No Scrubs");  // std::string
```

`auto` does not mean the variable has no type — C++ is still strictly typed. It just means the compiler fills in the type based on the initializer.

This is most useful when the type is long or obvious from context:

```cpp
std::vector<std::string> songs;
auto it = songs.begin();    // instead of std::vector<std::string>::iterator
```

`auto` only works when the compiler has enough information to deduce the type. If you declare a variable without initializing it, there is nothing for the compiler to work with:

```cpp
auto count;    // error: no initializer --- what type is this?
int count;     // OK: the type is explicitly int
```

Using `auto` for function parameters is allowed in C++20, but it turns the function into a template (a topic for a more advanced book). For now, spell out the type in function parameters:

```cpp
void print(auto x);    // valid C++20, but creates a template
void print(int x);     // clearer for now
```

And `auto` can pick a type you did not intend. For example, `auto volume = 11;` gives you an `int`, not a `short` or a `long` — if you need a specific type, spell it out.

You will see `auto` used more in later chapters when types get verbose. For simple declarations like `int count = 0`, spelling out the type is clearer.

Tip: Use `auto` when the type is obvious from the right-hand side or when it is painfully long to write out. Spell the type explicitly when the compiler cannot deduce the type or when the deduced type might not be what you want.

The sizeof Operator

The `sizeof` operator tells you how many bytes a type or variable occupies in memory. It has two forms:

```cpp
sizeof(type)
sizeof expression
```

When used with a type, parentheses are required. When used with a variable or expression, they are optional. `sizeof` returns a value of type `std::size_t`, which is an unsigned integer type.

```cpp
#include <iostream>

int main()
{
    std::cout << "char:      " << sizeof(char) << " bytes" << std::endl;
    std::cout << "int:       " << sizeof(int) << " bytes" << std::endl;
    std::cout << "double:    " << sizeof(double) << " bytes" << std::endl;
    std::cout << "long long: " << sizeof(long long) << " bytes" << std::endl;

    int score = 100;
    std::cout << "score:     " << sizeof(score) << " bytes" << std::endl;

    return 0;
}
```

On a typical 64-bit system, this might print:

```
char:      1 bytes
int:       4 bytes
double:    8 bytes
long long: 8 bytes
score:     4 bytes
```

`sizeof` is evaluated at compile time, not when the program runs. You can use it with a type name (like `sizeof(int)`) or with a variable (like `sizeof(score)`).

Wut: `sizeof(char)` is always 1, by definition. That does not mean a char is always 8 bits — it means the size of everything else is measured in multiples of `char`. On virtually all modern systems, a char is 8 bits, but the C++ standard does not require it.

std::numeric_limits

The `sizeof` operator tells you how many bytes a type occupies, but not what range of values it can hold. The `<limits>` header provides `std::numeric_limits<T>`, which lets you query the minimum and maximum values of any numeric type:

```
static constexpr T min();      // smallest value (integers) or smallest
                                // positive normalized value (floating-point)
static constexpr T max();      // largest value
static constexpr T lowest();   // most negative value
```

```
#include <iostream>
#include <limits>

int main()
{
    std::cout << "int min:    "
              << std::numeric_limits<int>::min()
              << std::endl;
    std::cout << "int max:    "
              << std::numeric_limits<int>::max()
              << std::endl;
    std::cout << "double max: "
              << std::numeric_limits<double>::max()
              << std::endl;

    return 0;
}
```

Wut: For floating-point types, `std::numeric_limits<double>::min()` is *not* the most negative `double` — it is the smallest *positive* normalized value (about 2.2e-308). If you want the most negative value, use `lowest()`. For integer types, `min()` and `lowest()` return the same value.

Arrays

An **array** is a collection of values of the same type, stored in contiguous memory. You declare an array by specifying its size in square brackets.

```
int scores[5] = {99, 85, 73, 91, 100};
```

Array indices start at 0, so `scores[0]` is 99 and `scores[4]` is 100.

```
std::cout << scores[0] << std::endl;   // prints 99
std::cout << scores[4] << std::endl;   // prints 100
```

Trap: Accessing an array out of bounds (like `scores[5]` in a 5-element array) is undefined behavior. C++ does not check array bounds for you — your program might crash, corrupt memory, or appear to work fine until it does not.

You can let the compiler figure out the size from the initializer:

```
int primes[] = {2, 3, 5, 7, 11};   // size is 5
```

To find the number of elements in an array, use `sizeof` on the array divided by `sizeof` one element:

```
int count = sizeof(primes) / sizeof(primes[0]);   // 5
```

The "value" of an array name is the address of its first element. This is important and will come up again in the Containers chapter.

Multidimensional Arrays

You can create arrays of arrays to represent grids, tables, or matrices.

```
int grid[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

`grid[0][0]` is 1, `grid[1][2]` is 7, and `grid[2][3]` is 12. The first index selects the row, the second selects the column.

You can think of `int grid[3][4]` as "3 rows of 4 ints each." The elements are laid out in memory row by row, so `grid[0][3]` and `grid[1][0]` are neighbors in memory.

Here is a program that prints a multiplication table using a 2D array:

```
#include <iostream>

int main()
{
    int table[5][5];

    for (int r = 0; r < 5; r++) {
        for (int c = 0; c < 5; c++) {
            table[r][c] = (r + 1) * (c + 1);
        }
    }

    for (int r = 0; r < 5; r++) {
        for (int c = 0; c < 5; c++) {
            std::cout << table[r][c] << "\t";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

Tip: The Containers chapter will introduce `std::array` and `std::vector`, which are safer and more flexible alternatives to raw arrays. Prefer those in modern C++ whenever possible.

const

The `const` keyword marks a variable as **read-only**. Once initialized, its value cannot be changed.

```
const double PI = 3.14159265358979;
const int MAX_LIVES = 3;
```

If you try to modify a `const` variable, the compiler will give you an error:

```
const int limit = 10;
limit = 20;   // ERROR: cannot assign to a const variable
```

Use `const` for values that should never change. It makes your intent clear to anyone reading your code and lets the compiler catch accidental modifications.

const with Pointers

When `const` meets pointers, things get interesting. We have not covered pointers in detail yet, but it is worth previewing this because it trips up many programmers.

There are two things that can be const: the **pointer itself** or the **data it points to**.

```
int vida = 99;

const int *p1 = &vida;     // pointer to const int
int *const p2 = &vida;     // const pointer to int
const int *const p3 = &vida; // const pointer to const int
```

With `const int *p1`, you cannot change the value through `p1` (`*p1 = 42` is an error), but you can make `p1` point somewhere else.

With `int *const p2`, you cannot make `p2` point somewhere else (`p2 = &other` is an error), but you can change the value through `p2` (`*p2 = 42` is fine).

With `const int *const p3`, you cannot do either.

Tip: Read pointer declarations from right to left. `const int *p` reads as "p is a pointer to int that is const" — the int is const. `int *const p` reads as "p is a const pointer to int" — the pointer is const.

Structures

A **structure** lets you group related data together under one name.

```
struct Song {
    std::string title;
    std::string artist;
    int year;
};
```

This defines a new type called `Song` with three **members**: `title`, `artist`, and `year`.

You access members using the **dot operator** (`.`):

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

struct Song {
    std::string title;
    std::string artist;
    int year;
};

int main()
{
    Song favorite;
    favorite.title = "Waterfalls";
    favorite.artist = "TLC";
    favorite.year = 1995;

    std::cout << favorite.title << " by " << favorite.artist
              << " (" << favorite.year << ")" << std::endl;

    return 0;
}
```

```
Waterfalls by TLC (1995)
```

You can also initialize a structure using curly braces:

```cpp
Song hit = {"No Scrubs", "TLC", 1999};
```

Structure Assignment

When you assign one structure to another, all members are **copied**:

```cpp
Song a = {"Livin' La Vida Loca", "Ricky Martin", 1999};
Song b = a;   // b is now a copy of a

b.year = 2000;

std::cout << a.year << std::endl;   // prints 1999 (unchanged)
std::cout << b.year << std::endl;   // prints 2000
```

Modifying `b` does not affect `a` because `b` has its own copy of all the data. This is an important detail — assignment copies the entire structure, member by member.

Wut: Structure assignment copies everything, which is usually what you want. But if the structure is very large, copying it can be expensive. We will revisit this when we talk about passing structures to functions.

Try It

Here is a program that puts several concepts from this chapter together. Type it in, compile it, and experiment with changes.

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

struct Cancion {
```

```
    std::string titulo;
    std::string artista;
    int anio;
};

int main()
{
    Cancion playlist[3] = {
        {"Waterfalls", "TLC", 1995},
        {"No Scrubs", "TLC", 1999},
        {"Livin' La Vida Loca", "Ricky Martin", 1999}
    };

    int count = sizeof(playlist) / sizeof(playlist[0]);

    for (int i = 0; i < count; i++) {
        std::cout << playlist[i].titulo << " - " << playlist[i].artista
                  << " (" << playlist[i].anio << ")" << std::endl;
    }

    return 0;
}
```

```
Waterfalls - TLC (1995)
No Scrubs - TLC (1999)
Livin' La Vida Loca - Ricky Martin (1999)
```

Key Points

- Every variable has a type that determines what it can hold and how much memory it uses.
- The fundamental types include `int`, `char`, `float`, `double`, `bool`, `short`, `long`, and their unsigned variants.
- A `char` is just a small integer; **ASCII** is the standard mapping between numbers 0–127 and the characters they represent. `std::cout` displays a `char` as a glyph and an `int` as digits, even when both hold the same value.
- Always initialize your variables — uninitialized variables contain garbage.
- `sizeof` tells you how many bytes a type or variable occupies.
- `std::numeric_limits<T>` from `<limits>` lets you query the min, max, and lowest values of any numeric type.
- Arrays store multiple values of the same type in contiguous memory, indexed starting at 0.
- Accessing an array out of bounds is undefined behavior — C++ will not catch it for you.
- `const` marks a value as read-only and makes your intent clear.
- With pointers, `const` can protect the pointer, the data, or both.
- Structures group related data together; assignment copies all members.

Exercises

1. On a system where `int` is 4 bytes, what is `sizeof(scores)` for `int scores[10]`?
2. What does the following program print?

```
#include <iostream>

int main()
{
    char c = 'C';
```

```
    c = c + 3;
    std::cout << c << std::endl;
    return 0;
}
```

3. What is wrong with the following code?

```
int data[3] = {10, 20, 30};
std::cout << data[3] << std::endl;
```

4. Consider the following declarations:

```
const int *p1 = nullptr;
int x = 42;
int *const p2 = &x;
```

Which one prevents you from changing the value being pointed to? Which one prevents you from changing where the pointer points?

5. What does the following program print?

```
#include <iostream>

struct Punto {
    int x;
    int y;
};

int main()
{
    Punto a = {3, 7};
    Punto b = a;
    b.x = 10;
    std::cout << a.x << " " << b.x << std::endl;
    return 0;
}
```

6. Why is it important to initialize variables before using them? What could happen if you read from an uninitialized `int`?

7. If `short` is 2 bytes, what is the maximum value an `unsigned short` can hold? How does this differ from a signed `short`?

8. Write a program that declares a structure to hold information about a car (make, model, year) and creates an array of 3 cars. Print out each car's information.

9. What does `std::numeric_limits<uint8_t>::max()` return? What about `std::numeric_limits<double>::min()` — is it a large negative number?

10. **What does this print?**

```
#include <iostream>

int main()
{
    auto a = 42;
    auto b = 42.0;
    auto c = 42 / 5;
    auto d = 42.0 / 5;
    std::cout << a << " " << b << " " << c << " " << d << "\n";
```

```
    return 0;
}
```

What is the deduced type of each variable?

11. **Calculation:** Given this declaration, what is the value at `grid[1][2]`?

```
int grid[3][4] = {
    {0, 1, 2, 3},
    {4, 5, 6, 7},
    {8, 9, 10, 11},
};
```

What is `sizeof(grid)` on a system where `int` is 4 bytes? How many `int` elements does `grid` hold in total?

12. **What does this print?**

```
#include <iostream>

int main()
{
    unsigned char x = 250;
    x = x + 10;
    std::cout << static_cast<int>(x) << "\n";
    return 0;
}
```

Why does `unsigned char` produce that result instead of `260`?

13. **What does this print?** Use the ASCII table to figure it out without running the code.

```
#include <iostream>

int main()
{
    char a = 'a';
    char b = a + 4;
    std::cout << b << " " << static_cast<int>(b) << std::endl;
    return 0;
}
```

3. Strings

In Chapter 2 you met the `char` type for single characters and saw that arrays can store sequences of values. You could, in principle, use a `char` array to store text. But raw `char` arrays are painful: you must track the length yourself, you cannot easily resize them, comparing two of them with `==` compares pointers rather than content, and a single missing null terminator can crash your program. `std::string` solves all of these problems — it manages its own memory, knows its own length, and provides a rich set of operations for searching, slicing, and combining text. In this chapter you will learn how to create strings, manipulate them, and convert between strings and numbers.

The `std::string` Type

To use `std::string`, you need to include the `<string>` header.

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

int main()
{
    std::string song = "MMMBop";
    std::cout << song << std::endl;
    return 0;
}
```

You can create strings in several ways:

```
std::string empty;                  // empty string ""
std::string greeting = "Hola";     // initialized with a string literal
std::string copy = greeting;        // copy of another string
std::string repeat(5, '!');         // "!!!!!" --- 5 copies of '!'
```

The first form creates an empty string, not an uninitialized one. This is different from how numeric types work — an uninitialized `int` contains garbage, but a default-constructed `std::string` is always `""`.

Tip: Always include `<string>` when using `std::string`. Some compilers let you get away without it because `<iostream>` may pull it in, but that is not guaranteed.

String Length

You can find out how many characters are in a string with `.size()` or `.length()`. Their signatures are:

```
size_t size() const;
size_t length() const;
```

They do exactly the same thing.

```
std::string title = "Ice Ice Baby";
std::cout << title.size() << std::endl;    // 12
std::cout << title.length() << std::endl;  // 12
```

An empty string has a size of 0. You can check if a string is empty with `.empty()`, which returns `true` or `false`. Its signature is:

```
bool empty() const;
```

```
std::string nada;
if (nada.empty()) {
```

```
    std::cout << "nothing here" << std::endl;
}
```

Concatenation

You can join strings together with the + operator. Its signature is:

```
std::string operator+(const std::string& lhs, const std::string& rhs);
```

This is called concatenation.

```
std::string first = "Baby";
std::string second = " One More Time";
std::string hit = first + second;
std::cout << hit << std::endl;  // Baby One More Time
```

You can also append to an existing string with +=. Its signature is:

```
std::string& operator+=(const std::string& str);
```

```
std::string lyrics = "Bailamos";
lyrics += ", te quiero";
std::cout << lyrics << std::endl;  // Bailamos, te quiero
```

You can concatenate a `std::string` with a string literal or a single character, but you cannot concatenate two string literals together with +.

```
std::string ok = std::string("ba ") + "da ba";  // works
// std::string bad = "ba " + "da ba";             // ERROR
```

Trap: The expression `"hello" + " world"` does not compile because both sides are string literals (character arrays), not `std::string` objects. At least one side of + must be a `std::string`.

Comparing Strings

You can compare strings using the familiar comparison operators: ==, !=, <, >, <=, >=. Their signatures follow this pattern:

```
bool operator==(const std::string& lhs, const std::string& rhs);
bool operator<(const std::string& lhs, const std::string& rhs);
// similarly for !=, >, <=, >=
```

Comparison is done character by character using the characters' numeric values (their ASCII codes).

```
std::string a = "Hanson";
std::string b = "Vanilla Ice";
if (a < b) {
    std::cout << a << " comes first" << std::endl;
}
```

This prints `Hanson comes first` because `'H'` (72) is less than `'V'` (86).

Be aware that uppercase letters have lower ASCII values than lowercase letters. So `"Zebra"` is less than `"apple"` because `'Z'` (90) is less than `'a'` (97).

Accessing Characters

You can access individual characters in a string using `[]` or `.at()`. Their signatures are:

```
char& operator[](size_t pos);
char& at(size_t pos);
```

Both use zero-based indexing, just like arrays (as we saw in Chapter 2).

```
std::string song = "MMMBop";
std::cout << song[0] << std::endl;     // M
std::cout << song.at(3) << std::endl;  // B
```

The difference is what happens when you go out of bounds. Using `[]` with an index greater than the string's length is undefined behavior — anything could happen. Accessing index `size()` with `[]` returns the null character `'\0'`, but anything beyond that is dangerous. Using `.at()` with an invalid index throws an exception that stops your program with a clear error message.

```
std::string word = "Hola";
// word[99]       --- undefined behavior, might crash, might not
// word.at(99)    --- throws std::out_of_range exception
```

Tip: Use `.at()` when you are not sure the index is valid. The small performance cost is worth the safety.

You can also modify individual characters this way.

```
std::string shout = "hey!";
shout[0] = 'H';
std::cout << shout << std::endl;  // Hey!
```

Iterating Through a String

You can loop through every character in a string using a range-based `for` loop.

```
std::string word = "Iris";
for (char c : word) {
    std::cout << c << ' ';
}
std::cout << std::endl;
// Output: I r i s
```

You can also use a traditional index-based loop.

```
std::string word = "Iris";
for (size_t i = 0; i < word.size(); ++i) {
    std::cout << word[i] << ' ';
}
std::cout << std::endl;
```

Tip: Use `size_t` (or `std::string::size_type`) for the loop variable when comparing against `.size()`. Using `int` can cause a signed/unsigned comparison warning from the compiler.

Finding and Extracting Substrings

The `.find()` method searches for a substring and returns the position where it was found. Its signatures are:

```
size_t find(const std::string& str, size_t pos = 0) const;
size_t find(char c, size_t pos = 0) const;
```

If the substring is not found, it returns `std::string::npos`.

```
std::string line = "Ice Ice Baby";
size_t pos = line.find("Baby");
if (pos != std::string::npos) {
    std::cout << "found at position " << pos << std::endl;  // 8
}
```

You can also search for a single character.

```
size_t space = line.find(' ');  // finds first space at position 3
```

The `.substr()` method extracts a portion of the string. Its signature is:

```
std::string substr(size_t pos = 0, size_t count = npos) const;
```

It takes a starting position and an optional length.

```
std::string song = "Baby One More Time";
std::string part = song.substr(5, 3);   // "One"
std::string rest = song.substr(9);      // "More Time"
```

If you omit the length, `.substr()` returns everything from the starting position to the end of the string.

Replacing Parts of a String

The `.replace()` method replaces a range of characters with new text. Its signature is:

```
std::string& replace(size_t pos, size_t count, const std::string& str);
```

You specify the starting position, the number of characters to replace, and the replacement string.

```
std::string phrase = "doo wop";
phrase.replace(0, 3, "bee bop");
std::cout << phrase << std::endl;  // bee bop wop
```

A common pattern is to combine `.find()` and `.replace()` to find and replace a specific substring.

```
std::string msg = "press play";
size_t pos = msg.find("play");
if (pos != std::string::npos) {
    msg.replace(pos, 4, "stop");
}
std::cout << msg << std::endl;  // press stop
```

Unicode and UTF-8

ASCII covers English letters, digits, and basic punctuation, but the world is much bigger than that. What about Spanish (¿, ñ, á)? Japanese (こんにちは)? Greek (Σωκράτης)? Emojis (🎵)? None of those characters have ASCII codes.

Unicode is the modern standard that gives every character in every writing system — plus a growing pile of emojis, mathematical symbols, and historical scripts — its own number, called a **code point**. Unicode reserves room for over 1.1 million code points, of which roughly 160,000 are currently assigned. A code point is just an integer the same way `'A'` is 65, except now the integers can run all the way up past a million.

That immediately raises a question: how do you store a code point that big in a `char`? A `char` only holds 256 values, so it cannot hold 160,000 — let alone 1,100,000 — directly. The trick is to use *several* `char`s for one Unicode character.

Several encodings exist for doing this; the most common one — and the default for C++ string literals — is **UTF-8**. The full rules of UTF-8 are out of scope for this book, but the part you need to know is simple:

- Every ASCII character (code points 0–127) is exactly **1 byte** in UTF-8, and that byte is identical to the ordinary ASCII byte. Old ASCII text is automatically valid UTF-8.
- Every other Unicode character takes **2, 3, or 4 bytes** in a row.

So a `std::string` can hold any Unicode text you want; under the hood it just stores the UTF-8 bytes and lets the terminal worry about drawing the glyphs.

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

int main()
{
    std::string spanish  = "¡Hola, mundo!";
    std::string japanese = "こんにちは";
    std::string emoji    = "🎵";

    std::cout << spanish  << " has " << spanish.size()  << " bytes\n";
    std::cout << japanese << " has " << japanese.size() << " bytes\n";
    std::cout << emoji    << " has " << emoji.size()    << " bytes\n";
    return 0;
}
```

Output:

```
¡Hola, mundo! has 14 bytes
こんにちは has 15 bytes
🎵 has 4 bytes
```

The Spanish string has 13 visible characters but 14 bytes, because ¡ is a 2-byte UTF-8 character and the rest are 1-byte ASCII. The Japanese string has 5 visible characters but 15 bytes, because each of those characters takes 3 bytes. The single musical note emoji takes 4 bytes all by itself.

Trap: `std::string::size()` returns the number of *bytes*, not the number of characters. For pure ASCII text the two are the same, but for any string containing non-ASCII characters they differ. There is no built-in way in standard C++ to count "characters" the way a human would — doing it correctly requires a Unicode library. For most string work — passing strings around, writing them to a file, sending them over a network — bytes are exactly what you want, so this is rarely a problem in practice.

Wut: Indexing into a `std::string` with `[]` or `.at()` gives you one *byte*, not one *character*. For an ASCII-only string they are the same thing, but `japanese[0]` from the example above gives you the first byte of こ, which is meaningless on its own. Slicing UTF-8 safely is another job for a Unicode library.

Writing Unicode in Source Code

If your editor cannot type a particular character, or you want to keep a source file pure ASCII, you can write a Unicode character using a hexadecimal escape. C++ has three flavors:

Escape	Meaning
`\xHH`	one byte with the given hex value
`\uHHHH`	Unicode code point, 4 hex digits
`\UHHHHHHHH`	Unicode code point, 8 hex digits

`\u` and `\U` take a *code point* and the compiler emits the right UTF-8 bytes for you. `\x` is lower-level: it places exactly that byte in the string, so you have to know what UTF-8 expects. For characters in the Basic Multilingual Plane (code points up to U+FFFF), `\u` is enough; for anything above that — including most emojis — you need `\U` with 8 digits.

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

int main()
{
    std::string spanish_x = "\xC2\xA1Hola!";   // ¡ as raw UTF-8 bytes
    std::string spanish_u = "\u00A1Hola!";     // ¡ via code point
    std::string note      = "\U0001F3B5";      // 🎵 via 32-bit code point

    std::cout << spanish_x << "\n";
    std::cout << spanish_u << "\n";
    std::cout << note      << "\n";
    return 0;
}
```

Output:

```
¡Hola!
¡Hola!
🎵
```

The first two strings produce identical output because `\xC2\xA1` *is* the UTF-8 encoding of code point U+00A1.

Trap: `\x` keeps eating hex digits as long as it sees them. `"\xC2A1"` is *not* the two bytes `0xC2 0xA1` — it is the single (oversized) hex value `0xC2A1`, which is an error. You can break the run by splitting the literal in two (`"\xC2" "A1"` — C++ glues adjacent string literals together) or just use `\u00A1` instead and skip the manual UTF-8.

String Input

As you saw in Chapter 1, `std::cin >> variable` reads input, but for strings it stops at the first whitespace character (space, tab, or newline).

```
std::string name;
std::cout << "enter your name: ";
std::cin >> name;
// If user types "Vanilla Ice", name is just "Vanilla"
```

To read an entire line of input, use `std::getline()`. Its signatures are:

```
std::istream& getline(std::istream& is, std::string& str);
std::istream& getline(std::istream& is, std::string& str, char delim);
```

```
std::string full_name;
std::cout << "enter your full name: ";
std::getline(std::cin, full_name);
// If user types "Vanilla Ice", full_name is "Vanilla Ice"
```

Trap: If you mix `std::cin >>` and `std::getline()`, you can run into trouble. After `std::cin >>` reads a value, the newline character from pressing Enter is left in the input buffer. The next `std::getline()` sees that newline and returns an empty string. Fix this by adding `std::cin.ignore()` between them. Its signature is:
`std::istream& ignore(std::streamsize count = 1, int_type delim = EOF);`

```
int age;
std::string name;
std::cout << "age: ";
std::cin >> age;
std::cin.ignore();              // discard the leftover newline
std::cout << "name: ";
std::getline(std::cin, name);   // now this works correctly
```

Converting Between Strings and Numbers

Sometimes you have a number stored as text and need to use it as an actual number, or vice versa.

To convert a string to a number, use `std::stoi()` (string to int) or `std::stod()` (string to double). Their signatures are:

```
int stoi(const std::string& str, size_t* pos = nullptr, int base = 10);
double stod(const std::string& str, size_t* pos = nullptr);
```

```
std::string year_str = "1997";
int year = std::stoi(year_str);
std::cout << year + 1 << std::endl;  // 1998

std::string price_str = "9.99";
double price = std::stod(price_str);
```

To convert a number to a string, use `std::to_string()`. Its signatures are:

```
std::string to_string(int value);
std::string to_string(double value);
// also overloaded for long, long long, unsigned, float, etc.
```

```
int track = 7;
std::string label = "Track " + std::to_string(track);
std::cout << label << std::endl;  // Track 7
```

Trap: If the string does not contain a valid number, `std::stoi()` and `std::stod()` throw an exception. For example, `std::stoi("abc")` will crash your program unless you handle the exception.

Try It

Here is a small program to experiment with. Try modifying it to use different string operations.

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

int main()
{
    std::string song = "Bailamos";
    std::cout << song << " has " << song.size()
```

```
              << " characters" << std::endl;

    song += ", mi amor";
    std::cout << song << std::endl;

    size_t pos = song.find("mi");
    if (pos != std::string::npos) {
        std::cout << "found 'mi' at position "
                  << pos << std::endl;
    }

    std::string piece = song.substr(0, 8);
    std::cout << "first word: " << piece << std::endl;

    return 0;
}
```

Key Points

- `std::string` from `<string>` manages text for you — no manual memory management needed.
- Use `.size()` or `.length()` to get the number of characters.
- Concatenate with `+` and `+=`, but at least one operand must be a `std::string`.
- Compare strings with `==`, `<`, `>`, etc. — comparison is character by character using ASCII values.
- Access characters with `[]` (fast, no bounds check) or `.at()` (safe, throws on bad index).
- Use `.find()` to search and `.substr()` to extract portions of a string.
- `std::cin >>` reads one word; `std::getline()` reads a whole line.
- Convert between strings and numbers with `std::stoi()`, `std::stod()`, and `std::to_string()`.
- A `std::string` holds **UTF-8** bytes, so any Unicode text fits, but `.size()` counts bytes (not characters) and indexing returns a single byte.

Exercises

1. What is the difference between `std::cin >> str` and `std::getline(std::cin, str)`? When would you use each one?
2. What does the following code print?

   ```
   std::string a = "Ice";
   std::string b = a + " " + a + " Baby";
   std::cout << b << std::endl;
   std::cout << b.size() << std::endl;
   ```

3. What is `std::string("Hola").at(4)`? What about `std::string("Hola")[4]`?
4. What is the value of `pos` after this code runs?

   ```
   std::string s = "MMMBop ba duba dop";
   size_t pos = s.find("dop");
   ```

5. Where is the bug in this code?

   ```
   std::string greeting = "Hello, " + "world!";
   std::cout << greeting << std::endl;
   ```

6. Where is the bug in this program?

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

```cpp
int main()
{
    int count;
    std::string name;
    std::cout << "how many? ";
    std::cin >> count;
    std::cout << "your name? ";
    std::getline(std::cin, name);
    std::cout << name << ": " << count << std::endl;
    return 0;
}
```

7. What does this code print?

```cpp
std::string s = "Bailamos";
for (char c : s) {
    if (c == 'a') {
        std::cout << '@';
    } else {
        std::cout << c;
    }
}
std::cout << std::endl;
```

8. If `std::stoi("42abc")` returns 42, what do you think `std::stoi("abc42")` does?

9. Write a program that asks the user for their full name using `std::getline()`, then prints:
   - the number of characters in their name
   - their name in reverse (print each character from last to first)

10. **What does this print?**

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

int main()
{
    std::string lyric = "Mmm bop, ba duba dop";
    lyric.replace(0, 3, "Pop");
    std::cout << lyric << "\n";
    return 0;
}
```

11. **What does this print?**

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

int main()
{
    std::string title = "Wannabe";
    std::cout << title.substr(0, 4) << "\n";
    std::cout << title.substr(3) << "\n";
    std::cout << title.substr(3, 100) << "\n";
    return 0;
}
```

The third call passes a length that runs off the end of the string. Does it crash, throw, or do something else?

12. **Think about it:** What does this print?

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

int main()
{
    std::string a = "Wonderwall";
    std::string b = "wonderwall";
    std::cout << (a == b) << "\n";
    std::cout << (a < b) << "\n";
    return 0;
}
```

String comparison is case-sensitive. Why is `a < b` true even though the words are spelled the same? What would you change to make the two strings compare equal regardless of case?

13. **What does this print?**

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

int main()
{
    std::string s = "café";
    std::cout << s << " " << s.size() << "\n";
    return 0;
}
```

`c`, `a`, and `f` are ASCII, but `é` is U+00E9, which UTF-8 encodes as 2 bytes. What does `s.size()` report, and why is it not 4?

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.

```
#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?

   ```
   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?

5. Control Flow

So far, every program you have written runs straight from top to bottom. Every line executes exactly once, in order. That is fine for simple tasks, but real programs need to make decisions and repeat actions. Control flow statements let your program choose which code to run and how many times to run it.

In this chapter, you will learn `if` statements for making decisions, loops for repeating code, and `switch` statements for selecting among multiple options. These are the building blocks that let your programs do interesting things.

`if` Statements

An `if` statement tests a condition and runs a block of code only when the condition is true:

```
#include <iostream>

int main() {
    int score = 85;

    if (score >= 90) {
        std::cout << "Excelente!\n";
    }

    return 0;
}
```

The condition inside the parentheses must be a boolean expression. If it evaluates to `true`, the code inside the braces runs. If it evaluates to `false`, the code is skipped entirely.

`else`

You can add an `else` block that runs when the condition is false:

```
int temperature = 30;

if (temperature > 100) {
    std::cout << "Too hot\n";
} else {
    std::cout << "Comfortable\n";
}
```

`else if`

When you have more than two possibilities, chain `else if` blocks together:

```
int score = 75;

if (score >= 90) {
    std::cout << "A\n";
} else if (score >= 80) {
    std::cout << "B\n";
} else if (score >= 70) {
    std::cout << "C\n";
} else {
    std::cout << "Try again\n";
}
```

The conditions are tested in order from top to bottom. As soon as one condition is true, its block runs and the rest are skipped. The final `else` catches anything that did not match an earlier condition.

Nested `if` Statements

You can put `if` statements inside other `if` statements:

```
int age = 20;
bool has_ticket = true;

if (age >= 18) {
    if (has_ticket) {
        std::cout << "Welcome to the show\n";
    } else {
        std::cout << "You need a ticket\n";
    }
} else {
    std::cout << "Must be 18 or older\n";
}
```

Guard Clauses

As you add more checks, nested `if` statements start drifting to the right side of the page and the real "success" path gets buried inside multiple pairs of braces. Imagine a function that has to check several things before letting someone into a show:

```
bool try_enter_show(int age, bool has_ticket, int seats_left) {
    if (age >= 18) {
        if (has_ticket) {
            if (seats_left > 0) {
                std::cout << "Welcome to the show\n";
                return true;
            } else {
                std::cout << "Sold out\n";
                return false;
            }
        } else {
            std::cout << "You need a ticket\n";
            return false;
        }
    } else {
        std::cout << "Must be 18 or older\n";
        return false;
    }
}
```

The success case ends up four levels deep, and each failure message sits far away from the condition that rejected it. Tracing why a particular message prints means counting open braces until you find the matching `else`.

A **guard clause** is an early `return` that rejects an invalid case as soon as you detect it, instead of wrapping the rest of the function in a `then` branch. Rewriting the same function with guard clauses:

```
bool try_enter_show(int age, bool has_ticket, int seats_left) {
    if (age < 18) {
        std::cout << "Must be 18 or older\n";
        return false;
```

```
    }
    if (!has_ticket) {
        std::cout << "You need a ticket\n";
        return false;
    }
    if (seats_left <= 0) {
        std::cout << "Sold out\n";
        return false;
    }
    std::cout << "Welcome to the show\n";
    return true;
}
```

Every failure case stands on its own at the same indentation level, the condition that triggers it sits right next to the message, and the happy path — the code that runs when *everything* checks out — lives at the bottom of the function, unindented. Readers no longer have to mentally stack up open braces to figure out which combination of conditions led to "Welcome to the show".

Tip: Deeply nested `if` statements make code hard to read. If you find yourself nesting more than two or three levels deep, consider using `else if` chains or guard clauses to flatten the logic.

Trap: A common mistake is using `=` (assignment) instead of `==` (comparison) in a condition:

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

The compiler may warn you about this. Compiling with `-Wall -Wextra` helps catch these mistakes.

`while` Loops

A `while` loop repeats a block of code as long as a condition is true. The condition is tested *before* each iteration, so if the condition is false from the start, the body never executes:

```
int countdown = 5;

while (countdown > 0) {
    std::cout << countdown << "... ";
    countdown--;
}
std::cout << "Vamos!\n";
// 5... 4... 3... 2... 1... Vamos!
```

Be careful to make sure the condition will eventually become false. If it never does, you have an infinite loop and your program will run forever (or until you press Ctrl+C).

Trap: Forgetting to update the loop variable is a classic source of infinite loops:

```
int i = 0;
while (i < 10) {
    std::cout << i << "\n";
    // oops, forgot i++ --- this runs forever
}
```

`do-while` Loops

A `do-while` loop is similar to `while`, but it tests the condition *after* the body. This guarantees the body executes at least once:

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

int main() {
    std::string input;

    do {
        std::cout << "Dime algo (or 'quit'): ";
        std::getline(std::cin, input);
        std::cout << "You said: " << input << "\n";
    } while (input != "quit");

    std::cout << "Adios!\n";
    return 0;
}
```

Notice the semicolon after `while (input != "quit")` — it is required for `do-while` and forgetting it is a syntax error.

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

Two keywords let you alter the normal flow inside a loop.

`break` exits the nearest enclosing loop immediately. `continue` skips the rest of the current iteration and jumps to the next iteration. In a `while` or `do-while` loop, that means the condition test. In a `for` loop, it jumps to the update expression (e.g., `i++`) first, and then the condition is tested.

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

int main() {
    // break example: stop searching when we find what we want
    std::string tracks[] = {
        "Losing My Religion",
        "Bitter Sweet Symphony",
        "Zombie"
    };

    for (int i = 0; i < 3; i++) {
        if (tracks[i] == "Zombie") {
            std::cout << "Found it at index " << i << "\n";
            break;
        }
    }

    // continue example: skip even numbers
```

```
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0)
            continue;
        std::cout << i << " ";
    }
    std::cout << "\n";
    // 1 3 5 7 9

    return 0;
}
```

`break` and `continue` only affect the innermost loop they appear in. If you have nested loops, a `break` in the inner loop does not exit the outer loop.

`for` Loops

The `for` loop is the most common loop in C++. It packs the initialization, condition, and update into a single line:

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

Here is a classic example:

```
for (int i = 0; i < 5; i++) {
    std::cout << i << " ";
}
std::cout << "\n";
// 0 1 2 3 4
```

The three parts of the `for` header work like this:

1. **init** runs once before the loop starts.
2. **condition** is tested before each iteration. If false, the loop ends.
3. **update** runs after each iteration, before the next condition test.

You can iterate over an array with a `for` loop using an index:

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

for (int i = 0; i < n; i++) {
    std::cout << "Score " << (i + 1) << ": " << scores[i] << "\n";
}
```

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

```
for (;;) {
    // runs forever --- use break to exit
}
```

Tip: Declare loop variables inside the `for` statement when possible: `for (int i = 0; ...)`. This limits the variable's scope to the loop body, preventing accidental use after the loop ends.

Range-Based `for` Loop

C++ provides a more convenient way to loop over collections called the range-based `for` loop:

```
int scores[] = {90, 84, 77, 95, 88};

for (int s : scores) {
    std::cout << s << " ";
}
std::cout << "\n";
// 90 84 77 95 88
```

The variable `s` takes on each value in `scores` one at a time. You do not need to manage an index variable or worry about going out of bounds.

This is just a brief introduction. You will use range-based `for` loops extensively in the Containers chapter, where they really shine with `std::vector` and `std::array`.

Wut: The range-based `for` loop makes a *copy* of each element by default. If you want to modify the elements in place, or avoid copying large objects, use a reference: `for (int &s : scores)`. We will cover references in more detail in the Functions chapter.

`switch` Statements

A `switch` statement selects among multiple cases based on the value of an integer or enumeration expression. It is useful when you are comparing one variable against several specific values:

```
#include <iostream>

int main() {
    int track = 2;

    switch (track) {
    case 1:
        std::cout << "Losing My Religion\n";
        break;
    case 2:
        std::cout << "Bitter Sweet Symphony\n";
        break;
    case 3:
        std::cout << "Zombie\n";
        break;
    default:
        std::cout << "Unknown track\n";
        break;
    }

    return 0;
}
```

Each `case` label must be a compile-time constant — you cannot use variables or strings as case labels.

Fall-Through

The most important thing to understand about `switch` is **fall-through** behavior. If you forget a `break`, execution continues into the next case. Sometimes this is intentional:

```cpp
char grade = 'B';

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

Here, cases `'A'`, `'B'`, and `'C'` all fall through to the same output. This is a common and useful pattern.

But accidental fall-through is a frequent bug:

```cpp
switch (x) {
case 1:
    std::cout << "uno\n";
    // oops, forgot break --- falls into case 2
case 2:
    std::cout << "dos\n";
    break;
}
```

If `x` is `1`, this prints both "uno" and "dos."

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. C++17 introduced the `[[fallthrough]]` attribute to make this intent explicit and silence compiler warnings.

The `default` Case

The `default` case runs when no other case matches. It is optional, but good practice to always include one — it catches unexpected values and makes your intent clear.

Try It: Control Flow Starter

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

int main() {
    // if / else
    int volume = 11;
    if (volume > 10) {
        std::cout << "It's a bitter sweet symphony, this life\n";
    } else {
        std::cout << "Turn it up\n";
    }
```

```cpp
// while loop
int n = 5;
while (n > 0) {
    std::cout << n << " ";
    n--;
}
std::cout << "Go!\n";

// do-while: keep asking until they say the magic word
std::string answer;
do {
    std::cout << "What's in your head? ";
    std::getline(std::cin, answer);
} while (answer != "zombie");
std::cout << "Zombie, zombie, zombie-ie-ie\n";

// for with break and continue
std::cout << "Odd numbers under 20: ";
for (int i = 1; i <= 100; i++) {
    if (i >= 20)
        break;
    if (i % 2 == 0)
        continue;
    std::cout << i << " ";
}
std::cout << "\n";

// range-based for
std::string playlist[] = {
    "Losing My Religion",
    "Bitter Sweet Symphony",
    "Zombie"
};
for (const std::string &song : playlist) {
    std::cout << "Now playing: " << song << "\n";
}

// switch
int choice = 2;
switch (choice) {
case 1:
    std::cout << "R.E.M.\n";
    break;
case 2:
    std::cout << "The Verve\n";
    break;
case 3:
    std::cout << "The Cranberries\n";
    break;
default:
    std::cout << "Unknown band\n";
    break;
}
```

```
    return 0;
}
```

Key Points

- `if`, `else if`, and `else` let your program make decisions based on conditions.
- `while` loops test the condition before each iteration. If the condition is false initially, the body never runs.
- `do-while` loops test the condition after each iteration, guaranteeing at least one execution of the body.
- `break` exits the nearest loop or `switch`. `continue` skips to the next iteration.
- `for` loops combine initialization, condition, and update in one line. Declare loop variables in the `for` header to limit their scope.
- Range-based `for` loops provide a cleaner way to iterate over arrays and containers.
- `switch` selects among cases using an integer or enum value. Watch for accidental fall-through — always use `break` unless fall-through is intentional.

Exercises

1. **Think about it:** When would you choose a `do-while` loop over a `while` loop? Describe a scenario where `do-while` is clearly the better choice and explain why.
2. **What does this print?**

   ```
   for (int i = 0; i < 5; i++) {
       if (i == 3)
           continue;
       std::cout << i << " ";
   }
   std::cout << "\n";
   ```

3. **What does this print?**

   ```
   int x = 2;
   switch (x) {
   case 1:
       std::cout << "uno ";
   case 2:
       std::cout << "dos ";
   case 3:
       std::cout << "tres ";
       break;
   default:
       std::cout << "other ";
   }
   std::cout << "\n";
   ```

4. **Where is the bug?**

   ```
   int i;
   int total = 0;
   for (i = 0; i < 10; i++);
   {
       total += i;
   }
   std::cout << "Total: " << total << "\n";
   ```

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. **What does this print?**

```
for (int i = 1; i <= 20; i++) {
    if (i % 3 == 0 && i % 5 == 0) {
        std::cout << "both ";
    } else if (i % 3 == 0) {
        std::cout << "tres ";
    } else if (i % 5 == 0) {
        std::cout << "cinco ";
    }
}
std::cout << "\n";
```

7. **Where is the bug?**

```
int n = 0;
while (n != 10) {
    std::cout << n << " ";
    n += 3;
}
```

8. **Write a program** that asks the user for a number between 1 and 7 and prints the day of the week using a `switch` statement. If the number is out of range, print an error message. Use a `do-while` loop to keep asking until the user enters `0` to quit.

9. **What does this print?**

```
#include <iostream>

int main()
{
    for (int row = 1; row <= 3; ++row) {
        for (int col = 1; col <= row; ++col) {
            std::cout << "*";
        }
        std::cout << "\n";
    }
    return 0;
}
```

Then change the inner loop to `for (int col = 1; col <= 4 - row; ++col)` and predict the new output.

10. **What does this print?**

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<std::string> tracks = {"Wonderwall", "Creep", "Linger"};
```

```
    for (const std::string &track : tracks) {
        std::cout << "- " << track << "\n";
    }
    return 0;
}
```

Why does the loop variable use `const std::string &` instead of just `std::string`?

11. **Write a program** that uses `break` to find the first negative number in an array. Use the array `int values[] = {3, 7, 2, -5, 4, -1};`. Print the index and value of the first negative number you find. If no negative number is found, print `"none"`.

12. **Think about it:** When would you intentionally let a `switch` case fall through into the next case, and how do you tell the compiler the fall-through is intentional rather than an accidental missing `break`?

6. Functions

As your programs grow, stuffing everything into `main` becomes unmanageable. Functions let you break a program into smaller, reusable pieces. Each function has a name, takes some input, does some work, and optionally returns a result.

You have already been using functions without thinking about it — `std::cout <<` calls operator functions behind the scenes, and `main` is itself a function. In this chapter you will learn how to write your own functions, how C++ passes data to and from them, and some powerful features like overloading and function pointers.

Declarations vs. Definitions

A function **declaration** (also called a prototype) tells the compiler the function's name, return type, and parameter types. A function **definition** provides the actual body — the code that runs when the function is called.

```
// Declaration (prototype) --- no body, just a semicolon
int add(int a, int b);

// Definition --- includes the body
int add(int a, int b) {
    return a + b;
}
```

The declaration tells the compiler "this function exists and here is its signature." The definition tells the compiler "here is what the function actually does."

Forward Declarations

The compiler reads your source file from top to bottom. If you call a function before the compiler has seen its definition, it does not know what the function looks like and reports an error. A **forward declaration** solves this by placing the declaration above the call:

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

// Forward declaration
void greet(const std::string &name);

int main() {
    greet("Mack");
    return 0;
}

// Definition comes after main
void greet(const std::string &name) {
    std::cout << "Return of the " << name << "!\n";
}
```

Without the forward declaration, the compiler would not know about `greet` when it encounters the call in `main`.

Tip: You can always avoid forward declarations by placing function definitions above where they are called. However, forward declarations are essential in larger programs where functions call each other, and they are the foundation of header files.

Header Files and the One-Definition Rule

When your program grows beyond a single file, you split it into multiple `.cpp` files. Each `.cpp` file, after all its `#include` directives are expanded, is called a **translation unit**. The compiler compiles each translation unit independently, and the linker combines them into a single program.

C++ enforces the **one-definition rule** (ODR): a function (or variable) can be *defined* only once across all translation units. Declarations can appear as many times as you like, but there must be exactly one definition.

This is why forward declarations exist — they let you declare a function in a header file so multiple `.cpp` files can call it, while the definition lives in exactly one `.cpp` file.

But what if you want to put a small helper function's *definition* directly in a header file? If two `.cpp` files both include that header, the linker sees two definitions and reports an error.

The `inline` keyword solves this:

```
// helpers.h
inline int max_volume() {
    return 11;
}
```

`inline` tells the compiler "this definition may appear in multiple translation units — treat them as the same definition." Now any `.cpp` file can include `helpers.h` without causing a linker error.

Tip: If you define a function in a header file (outside of a class body), mark it `inline` to avoid ODR violations. When you learn about classes in Chapter 12, you will see that member functions defined inside the class body are implicitly `inline`.

Wut: `inline` does not mean "the compiler will inline this function" (i.e., paste its body at each call site). It means "this definition may appear in multiple translation units." Modern compilers decide on their own whether to inline a call, regardless of the keyword.

`static` Functions

Sometimes you want a helper function that is used inside one `.cpp` file but is not meant to be called from anywhere else — it is purely an implementation detail of that file. Imagine two `.cpp` files that each need their own small `clamp` helper, each with a range that makes sense for that file:

```
// audio.cpp
int clamp(int v) {
    return v < 0 ? 0 : (v > 100 ? 100 : v);    // volume: 0..100
}

void set_volume(int v) {
    int safe = clamp(v);
    // ... apply safe volume
}
```

```
// video.cpp
int clamp(int v) {
    return v < -90 ? -90 : (v > 90 ? 90 : v);  // tilt angle: -90..90
}

void tilt_camera(int v) {
    int safe = clamp(v);
```

```
    // ... tilt the camera
}
```

Each file has its own `clamp` with a range that makes sense for *that* file's purpose. The problem is that at link time the linker sees two functions both named `clamp`, both with the signature `int(int)`, and reports a duplicate definition error. Even if the linker somehow picked one, `audio.cpp`'s `clamp` would still be callable from `video.cpp`, which defeats the idea that each `clamp` is a private helper.

The fix is the `static` keyword:

```
// audio.cpp
static int clamp(int v) {
    return v < 0 ? 0 : (v > 100 ? 100 : v);
}

// video.cpp
static int clamp(int v) {
    return v < -90 ? -90 : (v > 90 ? 90 : v);
}
```

A `static` function has **internal linkage**: the linker only makes the function visible inside the translation unit where it is defined. Each `.cpp` file gets its own private copy of `clamp`, so the two no longer collide, and nothing outside `audio.cpp` can accidentally call `audio.cpp`'s version — that version does not exist as far as the linker is concerned.

Use `static` on any file-scope function that is a helper for the current `.cpp` file and should not be part of the file's public interface. It is the C++ way of saying "private to this file."

Wut: The `static` keyword has another, completely unrelated meaning when applied to a variable *inside* a function: it makes that variable persist across calls instead of being recreated each time. That is local `static`, and it has nothing to do with linkage — it is just a name collision in the grammar.

Tip: Modern C++ offers a second way to get the same "private to this file" effect — put the helper inside an **anonymous namespace**:

```
namespace {
    int clamp(int v) { /* ... */ }
}
```

Anonymous namespaces apply to functions, variables, and types, while file-scope `static` only applies to functions and variables. For a single helper function, either style is fine.

`extern` Declarations

The mirror image of `static` is `extern`: it tells the compiler "the thing I am naming here is *defined* in another translation unit — trust me that the linker will find it."

You have already been using an implicit form of `extern` without knowing it. Every **function declaration** (the forward-declaration syntax you saw earlier in this chapter) is already `extern` by default:

```
int max_volume();          // declaration; implicitly extern
extern int max_volume();   // exact same thing, extern just made explicit
```

For functions you almost never write `extern` explicitly — just use the plain forward declaration.

Where `extern` actually matters is **global variables**. Suppose you want a global `master_volume` that every file in the program can read:

```cpp
// audio.cpp
int master_volume = 5;

// video.cpp
int master_volume;   // BUG: this DEFINES a second master_volume
```

Writing `int master_volume;` in `video.cpp` does not reference the one from `audio.cpp` — it creates a brand-new variable with its own storage inside `video.cpp`. The linker now sees two `master_volumes`, an ODR violation, and refuses to link.

`extern` fixes the problem by turning that line into a *declaration* instead of a definition:

```cpp
// video.cpp
extern int master_volume;   // declaration only; storage lives in audio.cpp
```

Now `video.cpp` can read and write `master_volume`, the actual storage lives in `audio.cpp`, and there is exactly one `master_volume` across the whole program. The usual pattern is to put the `extern` declaration in a header that every file includes, with the real definition in exactly one `.cpp` file:

```cpp
// globals.h
extern int master_volume;

// audio.cpp
#include "globals.h"
int master_volume = 5;

// video.cpp
#include "globals.h"
// master_volume is now visible --- no new definition created
```

When you need `extern`:

- To share a global variable across `.cpp` files. Put the `extern` declaration in a header, and put the real definition (with an initializer) in exactly one `.cpp` file.

When you do not need `extern`:

- For function declarations — they are already `extern` by default, so `int clamp(int v);` and `extern int clamp(int v);` are identical. Just write the plain forward declaration.
- For global variables that are only used inside one `.cpp` file. Mark those `static` (or put them in an anonymous namespace) so they stay private to that file.
- For local variables inside a function. They live on the stack and have no linkage to begin with.

Trap: Forgetting `extern` on a shared-global declaration in a header is the classic way to accidentally define a fresh copy of the variable in every translation unit that includes that header. Everything compiles cleanly, but at link time you get a duplicate definition error — or, even worse, the program links and every file silently operates on its own private copy of what was supposed to be a shared global. Always write `extern` on global-variable declarations in headers.

Parameters and Return Values

A function can take zero or more parameters and return a single value. The return type appears before the function name:

```cpp
int multiply(int x, int y) {
    return x * y;
}
```

You call the function by passing arguments that match the parameter types:

```cpp
int result = multiply(6, 7);
std::cout << result << "\n";   // 42
```

`void` Functions

If a function does not return a value, its return type is `void`:

```cpp
void print_chorus() {
    std::cout << "I want it that way\n";
}
```

You can still use `return;` (with no value) inside a `void` function to exit early, but you are not required to.

```cpp
void check_age(int age) {
    if (age < 0) {
        std::cout << "Invalid age\n";
        return;   // exit early
    }
    std::cout << "Age: " << age << "\n";
}
```

Pass-by-Value

By default, C++ passes arguments **by value**. This means the function receives a *copy* of the argument, not the original. Modifying the parameter inside the function does not affect the caller's variable:

```cpp
#include <iostream>

void try_to_change(int x) {
    x = 999;   // modifies the local copy only
}

int main() {
    int num = 42;
    try_to_change(num);
    std::cout << num << "\n";   // still 42
    return 0;
}
```

The function `try_to_change` modified its own copy of `x`, but `num` in `main` was never touched.

Pass-by-Reference

If you want a function to modify the caller's variable, pass it **by reference** using `&`:

```cpp
#include <iostream>

void make_it_louder(int &volume) {
    volume = 11;   // modifies the original
}

int main() {
    int vol = 5;
    make_it_louder(vol);
    std::cout << "Volume: " << vol << "\n";   // 11
    return 0;
}
```

The & after the type means `volume` is a reference — an alias for the caller's variable, not a copy. Any changes to `volume` are changes to `vol`.

A classic use of pass-by-reference is a swap function:

```
#include <iostream>

void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(x, y);
    std::cout << x << " " << y << "\n";   // 20 10
    return 0;
}
```

Tip: Use pass-by-value for small types like `int`, `char`, and `double`. Use pass-by-reference for larger types like `std::string` and structures to avoid the cost of copying.

`const` Parameters

Sometimes you want to pass by reference for efficiency (to avoid copying) but you do not want the function to modify the argument. Mark the parameter `const`:

```
void print_song(const std::string &title) {
    std::cout << "Now playing: " << title << "\n";
    // title = "something else";   // ERROR: title is const
}
```

The `const` reference gives you the best of both worlds: no copy is made, and the compiler guarantees the function cannot modify the original.

Tip: A good rule of thumb: if a function does not need to modify a parameter, make it `const`. For small types like `int`, pass by value. For larger types like `std::string`, pass by `const` reference. This communicates your intent clearly and lets the compiler catch accidental modifications.

Wut: A `const` reference can bind to a temporary value, but a non-const reference cannot:

```
void print_song(const std::string &title);
void modify_song(std::string &title);

print_song("Semi-Charmed Life");   // OK: const ref binds to temporary
// ERROR: non-const ref cannot bind to temporary
modify_song("Semi-Charmed Life");
```

This is why `const` references are so useful for function parameters that only read their argument.

Structures and Pass-by-Value

When you pass a structure by value, the entire structure is copied. For small structures this is fine, but for large ones the copy can be expensive:

```cpp
struct Album {
    std::string title;
    std::string artist;
    int year;
    int tracks;
};

// BAD: copies the entire Album struct
void print_album_bad(Album a) {
    std::cout << a.title << " by " << a.artist << "\n";
}

// GOOD: passes a const reference --- no copy, no modification
void print_album(const Album &a) {
    std::cout << a.title << " by " << a.artist << "\n";
}
```

Tip: Pass structures by `const` reference unless the function needs its own copy to modify. Copying a structure that contains strings or vectors copies all that data, which can be surprisingly slow.

Default Parameters

You can give function parameters default values. If the caller does not provide an argument for that parameter, the default is used:

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

void play(const std::string &song, int volume = 5) {
    std::cout << "Playing " << song << " at volume " << volume << "\n";
}

int main() {
    play("Return of the Mack");         // volume defaults to 5
    play("Return of the Mack", 11);     // volume is 11
    return 0;
}
```

Default parameters must appear at the end of the parameter list. You cannot put a defaulted parameter before a non-defaulted one:

```cpp
// OK
void play(const std::string &song, int volume = 5);

// ERROR: default parameter not at the end
void play(int volume = 5, const std::string &song);
```

Trap: Default values are specified in the declaration, not the definition (if they are separate). Specifying defaults in both places is an error:

```
// declaration with default
void play(const std::string &song, int volume = 5);

// definition --- no default here
void play(const std::string &song, int volume) {
    std::cout << song << " at " << volume << "\n";
}
```

Function Overloading

C++ lets you define multiple functions with the same name as long as their parameter lists differ. This is called **function overloading**:

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

void display(int value) {
    std::cout << "Integer: " << value << "\n";
}

void display(const std::string &value) {
    std::cout << "String: " << value << "\n";
}

void display(double value) {
    std::cout << "Double: " << value << "\n";
}

int main() {
    display(42);
    display("I want it that way");
    display(3.14);
    return 0;
}
```

The compiler decides which version to call based on the types of the arguments you pass. The functions must differ in the number or types of parameters — you cannot overload based on return type alone.

Recursive Functions

A function that calls itself is called a **recursive function**. Recursion is a powerful technique for problems that can be broken down into smaller versions of themselves.

The classic example is computing a factorial. The factorial of `n` (written `n!`) is `n * (n-1) * (n-2) * ... * 1`:

```
#include <iostream>

int factorial(int n) {
    if (n <= 1) {
        return 1;         // base case
    }
    return n * factorial(n - 1);  // recursive case
```

```
}

int main() {
    std::cout << "5! = " << factorial(5) << "\n";   // 120
    return 0;
}
```

Every recursive function needs two things:

1. A **base case** — a condition that stops the recursion.
2. A **recursive case** — where the function calls itself with a "smaller" problem.

Without a base case, the function calls itself forever until the program crashes with a stack overflow.

Trap: Forgetting the base case is the most common recursion mistake. Always ask yourself: "Under what condition does this function *not* call itself?" If you cannot answer that clearly, you have an infinite recursion.

Let's trace through `factorial(3)` to see how it works:

```
factorial(3)
  → 3 * factorial(2)
        → 2 * factorial(1)
              → returns 1     (base case)
        → returns 2 * 1 = 2
  → returns 3 * 2 = 6
```

Function Pointers

In C++, functions have addresses in memory just like variables do. A **function pointer** stores the address of a function so you can call it indirectly.

Here is the basic syntax:

```
#include <iostream>

int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }

int main() {
    // Declare a function pointer
    int (*operation)(int, int);

    operation = add;
    std::cout << operation(10, 3) << "\n";   // 13

    operation = subtract;
    std::cout << operation(10, 3) << "\n";   // 7

    return 0;
}
```

The declaration `int (*operation)(int, int)` means: `operation` is a pointer to a function that takes two `int` parameters and returns an `int`. The parentheses around `*operation` are required — without them, you would be declaring a function that returns an `int *`.

Simplifying with `using`

The function pointer syntax is admittedly ugly. You can clean it up with a `using` alias:

```
using MathOp = int (*)(int, int);

MathOp operation = add;
std::cout << operation(10, 3) << "\n";
```

Callbacks

Function pointers are commonly used as **callbacks** — you pass a function to another function, which calls it at the right time:

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

void process_songs(const std::string songs[], int count,
                   void (*action)(const std::string &)) {
    for (int i = 0; i < count; i++) {
        action(songs[i]);
    }
}

void announce(const std::string &song) {
    std::cout << "Now playing: " << song << "\n";
}

void shout(const std::string &song) {
    std::cout << ">> " << song << "!! <<\n";
}

int main() {
    std::string playlist[] = {
        "I Want It That Way",
        "Return of the Mack",
        "Semi-Charmed Life"
    };

    std::cout << "--- Chill mode ---\n";
    process_songs(playlist, 3, announce);

    std::cout << "--- Fiesta mode ---\n";
    process_songs(playlist, 3, shout);

    return 0;
}
```

The function `process_songs` does not know or care what `action` does — it just calls it for each song. This lets you change the behavior by passing a different function, without modifying `process_songs` itself.

Tip: Function pointers are the C++ mechanism for callbacks, but modern C++ often uses lambdas and `std::function` instead. You will encounter function pointers in older code and C libraries, so it is important to understand them.

`[[nodiscard]]`

Sometimes a function's return value is the whole point of calling it — ignoring it is almost certainly a bug. The `[[nodiscard]]` attribute tells the compiler to warn the caller if they discard the return value:

```
[[nodiscard]] int find_track(const std::string &playlist);
```

If someone calls `find_track("90s Jams")` without using the result, the compiler produces a warning:

```
find_track("90s Jams");           // warning: ignoring return value of
                                   // function declared with 'nodiscard'
int pos = find_track("90s Jams"); // OK --- return value is used
```

C++20 lets you add a reason that appears in the warning:

```
[[nodiscard("track index needed for playback")]]
int find_track(const std::string &playlist);
```

Tip: Use `[[nodiscard]]` on functions where ignoring the return value is almost certainly a bug — error codes, computed results, or newly allocated resources.

Operator Functions

Whew, there is a lot to functions in C++, right? Well, there is more: operator functions. To some they are an abomination, but to others, especially those writing core C++ APIs, they are fun ways to expand the semantics of language operators in new and unimagined ways.

You have actually been using operator overloading since Chapter 1. The `<<` and `>>` operators are built into C++ for shifting the bits of an integer left and right (you saw this in Chapter 4). But visually they look like they are pushing data in a direction — and the iostream library used that intuition to overload them for stream I/O. The compiler only knows how to use `<<` and `>>` with integers. The iostream library defined operator functions that taught the compiler how to apply `<<` and `>>` to `std::ostream`, `std::istream`, and other types. Every time you write `std::cout << "hello"`, you are calling an operator function.

An **operator function** lets you do the same thing for your own types — define what an operator like `+`, `==`, or `<<` does when applied to them. You write it as a regular function named `operator` followed by the symbol.

Here is a `Score` struct with overloaded `+`, `==`, and `>`:

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

struct Score {
    std::string player;
    int points;
};

Score operator+(const Score &a, const Score &b) {
    return Score{a.player + " & " + b.player, a.points + b.points};
}

bool operator==(const Score &a, const Score &b) {
    return a.points == b.points;
}

bool operator>(const Score &a, const Score &b) {
```

```
    return a.points > b.points;
}
```

Now you can use these operators naturally:

```
int main()
{
    Score a{"Fly", 95};
    Score b{"Intergalactic", 88};

    Score combined = a + b;
    std::cout << combined.player << ": " << combined.points << std::endl;

    if (a > b) {
        std::cout << a.player << " wins" << std::endl;
    }

    return 0;
}
```

Output:

```
Fly & Intergalactic: 183
Fly wins
```

The compiler sees `a + b` and looks for an `operator+` that takes two `Score` parameters. It finds yours and calls it like any other function.

Rules

There are a few rules that the compiler enforces:

- **You cannot create new operators.** You can overload `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `<<`, `>>`, `()`, `[]`, and many others, but you cannot invent `operator@` or `operator**`.
- **You cannot change arity** (the number of operands an operator takes). Binary operators stay binary (two operands), unary operators stay unary (one operand). You cannot make `+` take three operands.
- **You cannot change precedence or associativity.** Even if you overload `+` and `*`, the compiler still evaluates `*` before `+` just like it does with built-in types.
- **At least one operand must be a user-defined type.** You cannot redefine what `+` means for two `int`s.
- **Some operators cannot be overloaded at all:** `::`, `.`, `.*`, `?:`, and `sizeof`.

The << Operator for Output

You can overload `<<` so that `std::cout` works with your types. The left operand is an `std::ostream &` and the right operand is your type:

```
std::ostream &operator<<(std::ostream &os, const Score &s) {
    os << s.player << ": " << s.points;
    return os;
}
```

Returning the `os` reference is what lets you chain calls: `std::cout << a << " vs " << b`.

```
Score a{"Fly", 95};
std::cout << a << std::endl;  // prints: Fly: 95
```

Tip: Make your operators behave the way people expect. `+` should combine things, `==` should compare them, `<<` should print them. If `a + b` deleted `b`, your coworkers would not be amused.

Trap: Do not overload `&&`, `||`, or `,`. The built-in versions of `&&` and `||` use **short-circuit evaluation** — the right side is only evaluated if the left side does not already determine the result. When you overload them, both sides are *always* evaluated because the compiler treats them as regular function calls. This can break logic that depends on short-circuiting, like `ptr != nullptr && ptr->valid()`.

In Chapter 12 you will learn how to write operator functions as **member functions** of a class, which gives them access to private data. The operator functions shown here are free functions that work with structs whose members are public.

Try It: Functions Starter

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

// Forward declaration
void play(const std::string &song, int volume = 5);

// Pass by value vs pass by reference
void double_value(int x) {
    x *= 2;   // only modifies the copy
}

void double_ref(int &x) {
    x *= 2;   // modifies the original
}

// Recursive countdown
void countdown(int n) {
    if (n <= 0) {
        std::cout << "Vamos!\n";
        return;
    }
    std::cout << n << "... ";
    countdown(n - 1);
}

// Function pointer
using Formatter = void (*)(const std::string &);

void upper_announce(const std::string &s) {
    std::cout << ">> " << s << " <<\n";
}

void quiet_announce(const std::string &s) {
    std::cout << "(" << s << ")\n";
}

// Definition of play
```

```cpp
void play(const std::string &song, int volume) {
    std::cout << "Playing " << song << " at volume " << volume << "\n";
}

int main() {
    // Default parameters
    play("Semi-Charmed Life");
    play("Semi-Charmed Life", 11);

    // Pass by value vs reference
    int num = 10;
    double_value(num);
    std::cout << "After double_value: " << num << "\n";   // 10
    double_ref(num);
    std::cout << "After double_ref: " << num << "\n";     // 20

    // Recursion
    countdown(5);

    // Function pointer
    Formatter fmt = upper_announce;
    fmt("Return of the Mack");
    fmt = quiet_announce;
    fmt("Return of the Mack");

    return 0;
}
```

Key Points

- A function declaration tells the compiler about a function's signature. A definition provides the body.
- Forward declarations let you call a function before its definition appears in the file.
- The one-definition rule (ODR) requires exactly one definition of each function across all translation units. Use `inline` for functions defined in header files.
- C++ passes arguments by value by default — the function gets a copy.
- Use `&` to pass by reference when you need the function to modify the caller's variable or to avoid copying large objects.
- Mark parameters `const` when the function should not modify them. Use `const` references for large types you only need to read.
- Default parameters provide fallback values and must appear at the end of the parameter list.
- Function overloading lets you define multiple functions with the same name but different parameter lists.
- Recursive functions call themselves. Always ensure there is a base case to stop the recursion.
- Function pointers store the address of a function and enable callbacks — passing behavior as a parameter.
- Operator functions let you define what operators like `+`, `==`, and `<<` mean for your own types.
- You cannot create new operators, change arity, or change precedence.
- Do not overload `&&`, `||`, or `,` — it breaks short-circuit evaluation.
- `[[nodiscard]]` warns the caller if they ignore a function's return value.

Exercises

1. **Think about it:** Why does C++ pass arguments by value by default instead of by reference? What advantage does this give you in terms of reasoning about your code?

2. **What does this print?**

```
void mystery(int a, int &b) {
    a = a + 10;
    b = b + 10;
}

int main() {
    int x = 5, y = 5;
    mystery(x, y);
    std::cout << x << " " << y << "\n";
    return 0;
}
```

3. **Calculation:** What does `factorial(6)` return, using the recursive factorial function shown in this chapter?

4. **Where is the bug?**

```
int countdown(int n) {
    return n + countdown(n - 1);
}
```

5. **What does this print?**

```
void greet(const std::string &name) {
    std::cout << "Hola, " << name << "\n";
}

void greet(const std::string &name, int times) {
    for (int i = 0; i < times; i++) {
        std::cout << "Hola, " << name << "! ";
    }
    std::cout << "\n";
}

int main() {
    greet("Mack");
    greet("Mack", 3);
    return 0;
}
```

6. **Where is the bug?**

```
void set_volume(int volume = 5, const std::string &song) {
    std::cout << song << " at " << volume << "\n";
}
```

7. **What does this print?**

```
int apply(int (*func)(int, int), int a, int b) {
    return func(a, b);
}

int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }

int main() {
    std::cout << apply(add, 3, 4) << "\n";
```

```
    std::cout << apply(mul, 3, 4) << "\n";
    return 0;
}
```

8. **What does this print?**

```
struct Volume {
    int level;
};

Volume operator+(const Volume &a, const Volume &b) {
    return Volume{a.level + b.level};
}

bool operator>(const Volume &a, const Volume &b) {
    return a.level > b.level;
}

int main() {
    Volume a{5};
    Volume b{6};
    Volume c = a + b;
    std::cout << c.level << std::endl;
    std::cout << (a > b) << std::endl;
    return 0;
}
```

9. **Think about it:** Why should you not overload `&&` and `||`? What behavior do the built-in versions have that overloaded versions lose?

10. **Write a program** that defines a function `is_even` that returns `true` if a number is even and `false` otherwise. Write a second function `count_if` that takes an array of integers, its size, and a function pointer to a predicate (a function that takes an `int` and returns `bool`). `count_if` should return how many elements satisfy the predicate. Test it by counting the even numbers in an array.

11. **Where is the bug?** A coworker has a `helpers.h` header file with the following function definition. Two `.cpp` files both `#include "helpers.h"`. The program compiles, but the linker reports a "multiple definition" error. What is the fix?

```
// helpers.h
int double_it(int n) {
    return n * 2;
}
```

12. **What does the compiler do** with the following code?

```
[[nodiscard]] int compute(int a, int b) {
    return a * b;
}

int main() {
    compute(6, 7);
    return 0;
}
```

13. **Think about it / where is the bug?** Two functions take the same `Album` struct, which has a `std::string title`, a `std::string artist`, and an `int year`. Both functions only need to *read* the album.

```cpp
void print_album(Album a) {
    std::cout << a.title << " by " << a.artist << "\n";
}

void print_album_ref(const Album &a) {
    std::cout << a.title << " by " << a.artist << "\n";
}
```

Both compile and produce the same output. Why is the second version preferred for a struct like `Album`? Would the same answer apply to a function that takes a single `int`? Why or why not?

7. 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 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 2 that the character `'A'` is actually stored as the ASCII code `65`. 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.

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);
    }
}
```

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);
}
```

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);
}
```

Strings and Numbers

Programs frequently need to convert between strings and numbers — reading user input, parsing files, or displaying results. C++ provides several functions for this, each with different strengths. When you see `"52"` and `52` they may appear to be the same number, but to C++, the former is a `std::string` and not a number at all while the latter is an integer number. If we want to use `"52"` as a number, we need to convert the string to an integer.

Strings to Integers

The `std::stoi` function (string-to-integer) converts a `std::string` to an `int`. Its signature is:

```
int stoi(const std::string& str, std::size_t* pos = nullptr, int base = 10);
```

Used in its simplest form:

```
int a = std::stoi("42");        // 42
int b = std::stoi("  -7");      // -7 (leading whitespace is skipped)
int c = std::stoi("1984abc");   // 1984 (stops at first non-digit)
```

For larger values, use `std::stol` for `long` or `std::stoll` for `long long`:

```
// too large for int on most systems
long big = std::stol("3000000000");
long long huge = std::stoll("9000000000000");  // needs long long
```

Tip: `std::stoi` and its siblings throw `std::invalid_argument` if no conversion can be performed and `std::out_of_range` if the value won't fit. You will learn how to catch these exceptions in Chapter 11. Always be prepared to handle them when converting user input.

The `pos` Parameter

The second parameter, `pos`, receives the index of the first character that was **not** part of the number. This is useful when a string contains a number followed by other data:

```
std::size_t pos;
int val = std::stoi("42px", &pos);
// val == 42, pos == 2 (index of 'p')
```

You can use `pos` to parse multiple numbers from a single string or to check whether the entire string was consumed:

```
std::string input = "100 200 300";
std::size_t pos = 0;
int first = std::stoi(input, &pos);              // first == 100, pos == 3
int second = std::stoi(input.substr(pos), &pos); // second == 200
```

If you don't need `pos`, pass `nullptr` (or just omit it):

```
int x = std::stoi("42", nullptr);   // same as std::stoi("42")
```

Converting Bases with `std::stoi`

The third parameter of `std::stoi` specifies the base to use when parsing. You can use any base from 2 to 36:

```
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 same base parameter works with `std::stol` and `std::stoll`:

```
long d = std::stol("7C1", nullptr, 16);    // hex -> 1985
```

Tip: Pass base `0` and `std::stoi` will auto-detect the base from the prefix — `0x` for hex, `0b` for binary, and a leading `0` for octal:

```
std::stoi("0x2A", nullptr, 0);      // hex -> 42
std::stoi("0b101010", nullptr, 0);  // binary -> 42
std::stoi("052", nullptr, 0);       // octal -> 42
```

This is convenient, but beware: with base 0, the string `"010"` is parsed as **octal 8**, not decimal 10. If your data might have leading zeros that should be treated as decimal, specify base 10 explicitly.

Strings to Floating Point

The functions `std::stof`, `std::stod`, and `std::stold` convert strings to `float`, `double`, and `long double` respectively:

```
float  f = std::stof("3.14");       // 3.14f
double d = std::stod("2.71828");    // 2.71828
double e = std::stod("1.5e3");      // 1500.0 (scientific notation)
```

These follow the same pattern as the integer functions — they skip leading whitespace, stop at the first character that doesn't fit the number format, and throw the same exceptions for bad input.

Numbers to Strings

The `std::to_string` function converts numeric types back to strings:

```
std::string s1 = std::to_string(42);       // "42"
std::string s2 = std::to_string(-7);       // "-7"
std::string s3 = std::to_string(3.14);     // "3.140000"
```

For more control over formatting, use `std::format` (covered in Chapter 10):

```
std::string s = std::format("{:.2f}", 3.14);   // "3.14"
std::string h = std::format("{:#x}", 255);     // "0xff"
```

Manual Place-Value Math

You can also convert a number from one base to another by hand using place-value arithmetic — no library functions needed. To convert from another base to decimal, multiply each digit by its place value and add:

```
Hex "2A" to decimal:
  2 * 16^1 = 32
  A * 16^0 = 10
             --
             42

Binary "101010" to decimal:
  1*32 + 0*16 + 1*8 + 0*4 + 1*2 + 0*1 = 42
```

You can write a loop to do this programmatically using Horner's method — process each digit left to right, multiplying the running total by the base before adding the next digit:

```
std::string hex = "2A";
int result = 0;
for (char c : hex) {
    int digit = (c >= 'A') ? (c - 'A' + 10) : (c - '0');
    result = result * 16 + digit;
}
// result is 42
```

This is essentially what `std::stoi` does internally. Understanding the math helps you debug base-conversion issues and work with bases that library functions don't directly support.

Try It: Strings and Numbers

This program demonstrates conversions in both directions:

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

int main() {
    // string to integer
    int year = std::stoi("1985");
    std::println("Year: {}", year);

    // string to double
    double bpm = std::stod("150.2");
    std::println("BPM: {}", bpm);

    // number to string
    std::string msg = "Side " + std::to_string(1) + " of the cassette";
    std::println("{}", msg);

    // using the pos parameter
```

```
    std::size_t pos;
    int tempo = std::stoi("120bpm", &pos);
    std::println("Tempo: {} (unit starts at index {})", tempo, pos);

    // base conversions
    std::println("\n1985 in different bases:");
    std::println("  Hex \"7C1\"           = {}",
        std::stoi("7C1", nullptr, 16));
    std::println("  Binary \"11111000001\" = {}",
        std::stoi("11111000001", nullptr, 2));
    std::println("  Octal \"3701\"        = {}",
        std::stoi("3701", nullptr, 8));

    // base 0 auto-detection
    std::println("\nBase 0 auto-detection:");
    std::println("  \"0x7C1\"  = {}", std::stoi("0x7C1", nullptr, 0));
    std::println("  \"010\"    = {} (octal, not 10!)",
        std::stoi("010", nullptr, 0));
    std::println("  \"10\"     = {}", std::stoi("10", nullptr, 0));
}
```

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>

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
```

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

Type	Range
`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.

```
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. You will see `unsigned` again in the next chapter as `size_t`, the type returned by `.size()` on standard containers.

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 and Underflow?

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

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

```
unsigned char x = 255;
x = x + 1;  // x is now 0 (wraps around)
x = x - 2;  // x is now 254 (wraps around)
```

- **Signed overflow/underflow** 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());
}
```

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));
}
```

Bit Operators

We saw the logical operators in the previous chapter, in this chapter we will look at the version of those operators that operate on bits of a number.

Operator	Operation	Example (binary)	Result
`&`	AND	`0b1100 & 0b1010 | 0b1000 |`	
`|`	OR	`0b1100 | 0b1010 | 0b1110 |`	
`^`	XOR	`0b1100 ^ 0b1010 | 0b0110 |`	
`~`	NOT (complement)	`~0b1100`	flips all bits
`<<`	left shift	`0b0001 << 2 | 0b0100 |`	
`>>`	right shift	`0b1000 >> 2 | 0b0010 |`	

The `0b` prefix lets you write binary literals, which makes it easier to see what is happening bit by bit.

The bit operators work similarly to logical operators, except they work with the individual bits. When using the `&` operator a bit in the result will be `1` only if both corresponding bits are `1`. When using the `|` operator a bit in the result will be `1` only if either corresponding bits are `1`. There is also a XOR (`^`) operator that is true only if one or the other, but not both are true. Like, you can have fish or chicken but not both — you can have fish XOR chicken.

AND (`&`) is useful for checking if a specific bit is set.

```
int flags = 0b1010;
if (flags & 0b0010) {
    std::cout << "bit 1 is set" << std::endl;
}
```

OR (`|`) is useful for setting bits.

```
int flags = 0b1000;
flags |= 0b0010;   // flags is now 0b1010
```

XOR (`^`) flips bits — if a bit is 1, it becomes 0, and vice versa.

```
// flip bit 2 of a number
void toggle_bit_2(int &num) {
    num ^= 0b100;
}
```

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 in Chapter 1 — here we are talking about their original purpose as bitwise operators.

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);
    }
}
```

Key Points

- **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 parameter).
- **Converting between strings and numbers** is straightforward with `std::stoi`, `std::stod`, and `std::to_string`. The `pos` parameter tells you where parsing stopped, and the base parameter lets you parse hex, binary, and octal.
- **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!

Exercises

1. Convert the decimal number `200` to binary, hexadecimal, and octal by hand. Verify your answers by writing a C++ program that prints `200` in each base using `std::println`.
2. **What does this print?**

   ```
   int x = 0b1100;
   int y = 052;
   std::println("{}", x + y);
   ```

3. **Think about it:** In an 8-bit two's complement system, the most negative value is `-128` but the most positive value is only `127`. Why isn't the range symmetric?
4. **Where is the bug?** This loop is supposed to count down from 10 to 0, but it never terminates. Why?

   ```
   unsigned int count = 10;
   while (count >= 0) {
       std::println("{}", count);
       --count;
   }
   ```

5. Using two's complement with 8 bits, compute `100 - 75` by hand. Show the binary representation of `100`, the two's complement of `75`, and the binary addition.
6. **What values do `a`, `b`, and `c` hold** after these statements execute?

   ```
   int a = 1 << 10;
   int b = 100 >> 3;
   int c = (1 << 4) - 1;
   ```

7. **Where is the bug?** A programmer wrote this code and expected it to print `700`. What value does it actually print, and why?

```
int permissions = 0700;
std::println("Permissions: {}", permissions);
```

8. **What does this print?** What is wrong with this code?

```
int big = 2'000'000'000;
int doubled = big * 2;
std::println("{} * 2 = {}", big, doubled);
```

9. **Write a program** that reads a hexadecimal color code (like `"FF8000"`) from the user, converts it to its red, green, and blue components (each 0–255), and prints each component in decimal and binary. Use `std::stoi` with the base parameter and `substr` to extract each pair of hex digits.

10. **What does this print?**

```
uint8_t a = 250;
uint8_t b = 20;
uint8_t sum = a + b;
std::println("{} + {} = {}", a, b, sum);
```

11. **Write a program** to implement a function `bool is_set(int num, int bit)` that returns true if bit number `bit` of `num` is set. For example, if `is_set(13, 1)` is `false` but `is_set(13, 2)` is `true`.

12. **What does this print?**

```
int a = 1'000'000;
int b = 0xFF'00'FF;
int c = 0b1111'0000'1111'0000;
std::println("{} {} {}", a, b, c);
```

Are the digit separators part of the value? What does the program output?

13. **What does this print?**

```
std::string input = "42 100 255";
std::size_t pos = 0;

int a = std::stoi(input, &pos);
int b = std::stoi(input.substr(pos), &pos);
int c = std::stoi(input.substr(pos));

std::println("{} {} {}", a, b, c);
```

Walk through each call and explain what `pos` ends up as after each one.

14. **Calculation:** Without running a program, compute each of these in 8-bit binary, then give the result in decimal:

```
0b1010'1100 & 0b1111'0000
0b1010'1100 | 0b0000'1111
0b1010'1100 ^ 0b1111'1111
```

What does XORing with all-ones do?

15. **Calculation:** On a typical 64-bit Linux system, what does each of these print?

```
sizeof(char)
sizeof(short)
sizeof(int)
```

```
sizeof(long)
sizeof(long long)
```

For each type, what is the largest value an *unsigned* version of that type can hold? (You may answer in terms of `2^N - 1` instead of writing the full decimal number.)

16. **Write a program** that asks the user for an integer and prints it in decimal, hex, octal, binary, and again as a `std::string` produced by `std::to_string`. Use `std::println` (or `std::format`) for the formatted output.

8. Containers

In Chapter 2 you used C-style arrays declared with `[]` to store sequences of values, and in Chapters 5 and 6 you wrote loops and functions to work with them. But C-style arrays have serious problems: they do not know their own size, they silently decay to pointers when passed to functions (losing size information), they cannot be returned from functions, and they cannot grow or shrink at runtime. You end up passing a separate size parameter everywhere and hoping nothing goes out of bounds. The standard library provides **containers** — classes that manage collections of elements and solve all of these problems. They know their size, they can be passed to and returned from functions safely, and some can resize dynamically. In this chapter you will learn `std::array` for fixed-size collections, `std::vector` for dynamic collections, and how to iterate through both using range-based for loops and iterators.

std::array

`std::array` fixes the C-style array problems described above. It is a fixed-size container that wraps a C-style array and gives it a proper interface. To use it, include the `<array>` header.

Here are the key methods you will use most often:

```
T& operator[](size_t pos);          // access element, no bounds checking
// access element, throws if out of range
T& at(size_t pos);
constexpr size_t size() const;      // number of elements
```

These work the same way as the string methods you saw in Chapter 3, but now they operate on whatever type `T` the array holds.

```
#include <array>
#include <iostream>

int main()
{
    std::array<int, 5> scores = {90, 85, 92, 88, 76};

    std::cout << "First score: " << scores[0] << "\n";
    std::cout << "Number of scores: " << scores.size() << "\n";

    return 0;
}
```

```
First score: 90
Number of scores: 5
```

The declaration `std::array<int, 5>` creates an array that holds exactly 5 `int` values. The type and the size are both part of the type — a `std::array<int, 5>` is a different type from a `std::array<int, 10>`.

Accessing Elements

You can access elements in two ways:

```
std::array<std::string, 2> songs = {"Wannabe", "No Diggity"};

// Using [] --- no bounds checking, fast
std::cout << songs[0] << "\n";    // Wannabe

// Using .at() --- bounds checking, safer
std::cout << songs.at(1) << "\n"; // No Diggity
```

Tip: Use `.at()` while developing and debugging. It throws an exception if the index is out of range, which immediately tells you something is wrong. The `[]` operator does not check bounds — accessing an invalid index is undefined behavior.

Why std::array Over C-Style Arrays?

A `std::array` knows its own size. You can pass it to a function and the function knows how many elements it contains:

```
#include <array>
#include <iostream>

void print_scores(const std::array<int, 3>& scores)
{
    std::cout << "There are " << scores.size() << " scores\n";
    for (size_t i = 0; i < scores.size(); i++) {
        std::cout << "  " << scores[i] << "\n";
    }
}

int main()
{
    std::array<int, 3> my_scores = {95, 87, 91};
    print_scores(my_scores);
    return 0;
}
```

```
There are 3 scores
  95
  87
  91
```

With a C-style array you would have to pass the size as a separate parameter. With `std::array`, the size is built in.

Trap: The size of a `std::array` must be known at compile time. You cannot write `std::array<int, n>` where `n` is a variable. If you need a size determined at runtime, use `std::vector` instead.

std::vector

`std::vector` is the workhorse container of C++. It is a dynamic array that can grow and shrink as needed. To use it, include the `<vector>` header:

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> numbers = {10, 20, 30};

    std::cout << "Size: " << numbers.size() << "\n";
    std::cout << "First: " << numbers[0] << "\n";
```

```
    return 0;
}
```

```
Size: 3
First: 10
```

Creating Vectors

There are several ways to create a vector:

```
std::vector<int> empty;                    // empty vector
std::vector<int> zeros(5);                 // 5 elements, all 0
std::vector<int> fives(5, 42);             // 5 elements, all 42
// initializer list
std::vector<std::string> songs = {"Wannabe", "No Diggity"};
```

The **initializer list** syntax with {} is the most common way to create a vector with specific values.

Adding and Removing Elements

The power of `std::vector` is that it can grow. The two main methods for changing a vector's contents are:

```
void push_back(const T& value);  // add element to the end
void pop_back();                 // remove last element
```

```
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> playlist;

    playlist.push_back("Wannabe");
    playlist.push_back("No Diggity");

    std::cout << "Playlist has " << playlist.size() << " songs\n";

    playlist.pop_back();  // removes the last element

    std::cout << "After pop: " << playlist.size() << " songs\n";
    std::cout << "Remaining: " << playlist[0] << "\n";

    return 0;
}
```

```
Playlist has 2 songs
After pop: 1 song
Remaining: Wannabe
```

`push_back` adds an element to the end of the vector. `pop_back` removes the last element.

Trap: Calling `pop_back()` on an empty vector is undefined behavior. Always check that the vector is not empty first using `.empty()` or `.size()`.

Accessing Elements

Vectors provide multiple ways to access elements. Like `std::array`, vectors support `operator[]` and `.at()` for indexed access. They also provide methods for the first and last elements:

```
T& front();    // first element
T& back();     // last element
```

```
std::vector<std::string> bands = {"Spice Girls", "Blackstreet", "Oasis"};

std::cout << bands[0] << "\n";        // Spice Girls --- no bounds check
std::cout << bands.at(1) << "\n";     // Blackstreet --- bounds checked
std::cout << bands.front() << "\n";   // Spice Girls --- first element
std::cout << bands.back() << "\n";    // Oasis --- last element
```

Just like with `std::array`, `.at()` throws an exception on an invalid index while `[]` does not check.

Size, Capacity, and Empty

A vector tracks two things: its **size** (how many elements it holds) and its **capacity** (how much memory it has allocated). You already know `.size()` — the `.capacity()` method tells you how much room has been allocated:

```
// number of elements held without reallocating
size_t capacity() const;
```

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v;

    std::cout << "Size: " << v.size()
              << ", Capacity: " << v.capacity() << "\n";

    for (int i = 0; i < 5; i++) {
        v.push_back(i * 10);
        std::cout << "Size: " << v.size()
                  << ", Capacity: " << v.capacity() << "\n";
    }

    return 0;
}
```

The output will look something like:

```
Size: 0, Capacity: 0
Size: 1, Capacity: 1
Size: 2, Capacity: 2
Size: 3, Capacity: 4
Size: 4, Capacity: 4
Size: 5, Capacity: 8
```

Notice that capacity grows in jumps. When the vector runs out of room, it allocates a larger block of memory (typically doubling) and copies everything over. This means `push_back` is usually fast, but occasionally it has to do extra work.

The `.empty()` method returns `true` if the vector has no elements:

```
bool empty() const;  // true if size() == 0
```

```
if (v.empty()) {
    std::cout << "Nothing here\n";
}
```

To remove all elements, use `.clear()`:

```
void clear();  // remove all elements, size becomes 0
```

```
v.clear();
std::cout << "Size after clear: " << v.size() << "\n";  // 0
```

Wut: After calling `.clear()`, the size is 0 but the capacity is unchanged. The vector still holds onto its allocated memory. This is intentional — if you are going to refill it, there is no point in freeing the memory just to reallocate it.

Inserting, Erasing, and Reserving

`push_back` and `pop_back` work at the end of the vector. Sometimes you need to insert or remove elements at other positions.

The `.insert()` method inserts an element before a given position. The `.erase()` method removes an element (or a range of elements) at a given position. Both take a position expressed as an **iterator** — you will learn more about iterators in the next section, but for now, `vec.begin() + n` means "the position at index n":

```
iterator insert(const_iterator pos, const T& value);
iterator erase(const_iterator pos);
iterator erase(const_iterator first, const_iterator last);
```

```
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> lista = {"Creep", "No Rain", "Linger"};

    // Insert "Possum Kingdom" at index 1 (before "No Rain")
    lista.insert(lista.begin() + 1, "Possum Kingdom");

    // Erase the element at index 0 ("Creep")
    lista.erase(lista.begin());

    for (const auto& s : lista) {
        std::cout << s << "\n";
    }

    return 0;
}
```

```
Possum Kingdom
No Rain
Linger
```

Trap: Inserting or erasing elements can **invalidate** existing iterators, pointers, and references to elements in the vector. After an insert or erase, do not use any iterators you obtained before the operation — get fresh ones.

Inserting and erasing at positions other than the end are slower than `push_back` and `pop_back` because elements must be shifted in memory. If you need frequent insertion and removal in the middle, other containers may be more efficient.

Two methods let you manage capacity explicitly:

```
void reserve(size_t new_cap);   // preallocate memory without adding elements
// request that capacity be reduced to match size
void shrink_to_fit();
```

`reserve` is useful when you know how many elements you will add. It avoids the repeated reallocations that happen when a vector grows one element at a time:

```
std::vector<int> v;
v.reserve(1000);              // allocate room for 1000 ints
// v.size() is still 0, but v.capacity() is at least 1000
```

`shrink_to_fit` is a non-binding request — the implementation is allowed to ignore it.

You can also construct a vector from a range of iterators, copying elements from another container or a subrange:

```
std::vector<int> all = {10, 20, 30, 40, 50};
std::vector<int> middle(all.begin() + 1, all.begin() + 4);
// middle is {20, 30, 40}
```

Iterating Through Containers

Now that you have containers with data in them, you need to loop through them. C++ gives you several ways to do this.

Range-Based For Loop

The simplest and most modern way to iterate is the **range-based for loop**:

```
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> songs = {"Wannabe", "No Diggity"};

    for (const auto& song : songs) {
        std::cout << song << "\n";
    }

    return 0;
}
```

```
Wannabe
No Diggity
```

The syntax `for (const auto& song : songs)` means "for each element in `songs`, call it `song`." The `const auto&` part means:

- `auto` — let the compiler figure out the type
- `&` — use a reference so the element is not copied
- `const` — promise not to modify the element

If you want to modify the elements, drop the `const`:

```
std::vector<int> values = {1, 2, 3, 4, 5};

for (auto& v : values) {
    v *= 10;  // modify each element in place
}
// values is now {10, 20, 30, 40, 50}
```

Tip: Use `const auto&` when you only need to read elements. Use `auto&` when you need to modify them. Avoid plain `auto` (without `&`) for anything larger than a primitive type — it makes a copy of each element, which is wasteful.

Using Iterators

Under the hood, the range-based for loop uses **iterators**. An iterator is an object that points to an element in a container, similar to how a pointer points to a memory address.

Every container provides `.begin()` and `.end()`:

```
iterator begin();  // iterator to the first element
iterator end();    // iterator to one past the last element
```

- `.begin()` returns an iterator pointing to the first element
- `.end()` returns an iterator pointing to one past the last element

```
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> canciones = {"Wannabe", "No Diggity"};

    for (auto it = canciones.begin(); it != canciones.end(); ++it) {
        std::cout << *it << "\n";
    }

    return 0;
}
```

```
Wannabe
No Diggity
```

The `*it` syntax **dereferences** the iterator to get the element it points to, just like dereferencing a pointer. The `++it` moves the iterator to the next element.

Wut: `.end()` does not point to the last element — it points to one *past* the last element. This is a deliberate design choice in C++ called a "half-open range." It means the valid range is `[begin, end)` in mathematical notation. This makes loops cleaner and avoids off-by-one errors.

You might wonder why you would use iterators directly when the range-based for loop exists. Some algorithms in the standard library require iterators, and iterators give you more control when you need to do things like erase elements while iterating or iterate backward.

auto with Iterators

Without `auto`, iterator types can be verbose:

```
// Without auto --- quite a mouthful
std::vector<std::string>::iterator it = canciones.begin();

// With auto --- much cleaner
auto it = canciones.begin();
```

This is one of the best uses of `auto`. The type is obvious from context, and `auto` saves you from writing out the full iterator type.

Try It: Container Starter

Here is a program that exercises vectors and iteration. Type it in, compile it, and experiment:

```
#include <array>
#include <iostream>
#include <string>
#include <vector>

int main()
{
    // std::array
    std::array<int, 4> fixed = {10, 20, 30, 40};
    std::cout << "Array size: " << fixed.size() << "\n";
    std::cout << "Element 2: " << fixed.at(2) << "\n\n";

    // std::vector
    std::vector<std::string> lista;
    lista.push_back("Wannabe");
    lista.push_back("No Diggity");

    std::cout << "Playlist:\n";
    for (const auto& song : lista) {
        std::cout << "  " << song << "\n";
    }
    std::cout << "\n";

    // Modify elements
    std::vector<int> nums = {1, 2, 3, 4, 5};
    for (auto& n : nums) {
        n *= 2;
    }
    std::cout << "Doubled: ";
    for (const auto& n : nums) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    // Size vs. capacity
```

```
    std::vector<int> v;
    for (int i = 0; i < 8; i++) {
        v.push_back(i);
        std::cout << "size=" << v.size()
                  << " cap=" << v.capacity() << "\n";
    }

    return 0;
}
```

Key Points

- `std::array<T, N>` is a fixed-size container that knows its size and works with standard library algorithms. The size `N` must be a compile-time constant.
- `std::vector<T>` is a dynamic container that grows and shrinks as needed using `push_back()` and `pop_back()`.
- Use `.at()` for bounds-checked access and `[]` for unchecked access.
- Use `.front()` and `.back()` to access the first and last elements.
- A vector's **capacity** is the amount of memory it has allocated; its **size** is how many elements it actually holds.
- `insert()` and `erase()` modify a vector at any position but invalidate existing iterators. `reserve()` preallocates memory; `shrink_to_fit()` releases unused memory.
- The range-based for loop (`for (const auto& x : container)`) is the simplest way to iterate.
- Iterators provide lower-level access to container elements using `.begin()` and `.end()`.
- Use `auto` to avoid writing verbose iterator types.

Exercises

1. **Think about it:** Why does `std::array` require the size as part of its type (e.g., `std::array<int, 5>`) while `std::vector` does not? What trade-off does this create?

2. **What does this print?**

   ```
   std::vector<int> v = {10, 20, 30};
   v.push_back(40);
   v.pop_back();
   v.pop_back();
   std::cout << v.size() << " " << v.back() << "\n";
   ```

3. **Calculation:** If a `std::vector<int>` has a capacity of 8 and a size of 5, how many more elements can you `push_back` before it needs to reallocate memory?

4. **Where is the bug?**

   ```
   std::vector<int> scores;
   scores.push_back(95);
   scores.push_back(87);
   scores.push_back(91);

   for (int i = 0; i <= scores.size(); i++) {
       std::cout << scores[i] << "\n";
   }
   ```

5. **What does this print?**

   ```
   std::array<int, 4> a = {5, 10, 15, 20};
   for (auto it = a.begin(); it != a.end(); ++it) {
   ```

```
    std::cout << *it << " ";
}
std::cout << "\n";
```

6. **Think about it:** The range-based for loop `for (auto x : vec)` (without `&`) works, but why is it generally a bad idea for vectors of strings? When would it be acceptable?

7. **Where is the bug?**

```
std::vector<std::string> playlist = {"Wannabe", "No Diggity"};
std::cout << playlist.at(2) << "\n";
```

8. **Calculation:** A `std::vector<double>` contains 3 elements and has a capacity of 4. You call `push_back` 5 times. After all 5 calls, what is the size? Assuming the capacity doubles when exceeded, what is the capacity?

9. **What does this print?**

```
std::vector<int> v = {1, 2, 3};
v.clear();
std::cout << v.size() << " " << v.empty() << "\n";
```

10. **Write a program** that asks the user to enter numbers one at a time (enter -1 to stop), stores them in a `std::vector<int>`, and then prints them in reverse order using iterators or indexing.

11. **What does this print?**

```
std::vector<int> v = {10, 20, 30, 40, 50};
v.insert(v.begin() + 2, 25);
v.erase(v.begin());
for (const auto& n : v) {
    std::cout << n << " ";
}
std::cout << "\n";
```

12. **Calculation:** What is the size and capacity after calling `reserve(100)` on an empty `std::vector<int>`, then calling `push_back` 3 times?

13. **Where is the bug?**

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<int> v = {1, 2, 3, 4, 5};
    for (auto it = v.begin(); it != v.end(); ++it) {
        if (*it == 3) {
            v.push_back(99);
        }
    }
    for (int n : v) std::cout << n << " ";
    std::cout << "\n";
    return 0;
}
```

    The program may crash, may print garbage, or may even appear to work depending on the compiler. What is going on, and how would you fix it?

14. **Think about it:** A function takes three coordinates as a `std::array<double, 3>`:

```
double length(const std::array<double, 3> &v);
```

Why does the size `3` appear in the parameter type, and what would happen if a caller passed a `std::array<double, 4>` instead? Compare this to a function that takes `const std::vector<double> &v` — which one is more flexible, and which one gives stronger compile-time guarantees?

15. **Calculation:** Start with an empty `std::vector<int>` and call `reserve(8)`. Then call `push_back` 12 times. What are `size()` and `capacity()` after each of those 12 calls? (Assume the implementation doubles the capacity when it has to grow.) On which `push_back` calls (if any) are existing iterators into the vector invalidated?

9. I/O Streams

In Chapter 1 you met `std::cout` and `std::cin` for printing to the screen and reading from the keyboard. Every program you have written since then reads from the keyboard and writes to the console. But console I/O is ephemeral — once your program ends, everything it printed is gone. You cannot save results for later, you cannot process data from an existing file, and you cannot build a complex string in memory before outputting it. C++ solves this with a uniform streaming interface: the `<<` and `>>` operators you already know work the same way with files and in-memory strings as they do with the console. In this chapter you will learn about string streams for building and parsing strings in memory, and file streams for reading and writing files.

A Quick Review

You already know the basics from Chapter 1:

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

int main()
{
    std::string song;

    std::cout << "Favorite 90s song? ";
    std::getline(std::cin, song);
    std::cout << "Good choice: " << song << std::endl;

    return 0;
}
```

`std::cout` is an **output stream** that sends data to the screen. `std::cin` is an **input stream** that reads data from the keyboard. The `<<` operator inserts data into an output stream, and `>>` extracts data from an input stream.

What makes C++ streams powerful is that this same interface — `<<` and `>>` — works with string streams and file streams too. Once you learn one, you know them all.

Stream Manipulators

Before `std::format` arrived in C++20 (Chapter 10), C++ formatted output using **stream manipulators** — special values you insert into a stream with `<<` to change how subsequent output is formatted. They live in the `<iomanip>` and `<iostream>` headers. You will encounter them in older code, so it is worth knowing what they do.

Output manipulators:

On	Off	Description
`std::boolalpha`	`std::noboolalpha`	print `bool` as `true`/`false` instead of `1`/`0`
`std::fixed`	`std::defaultfloat`	fixed-point notation for floating-point numbers
`std::scientific`	`std::defaultfloat`	scientific notation (e.g., `3.14e+00`)
`std::showpoint`	`std::noshowpoint`	always show the decimal point
`std::showpos`	`std::noshowpos`	show `+` sign for positive numbers
`std::left`	—	left-align within field width
`std::right`	—	right-align within field width (default)
`std::setw(n)`	—	pad the next value to at least `n` characters; only applies to the next `<<`, then resets

On	Off	Description
`std::setprecision(n)`	—	set number of digits (significant, or decimal places with `std::fixed`)
`std::setfill(c)`	—	set the fill character (default is space)

`std::setw`, `std::setprecision`, and `std::setfill` require the `<iomanip>` header. The rest are in `<iostream>`.

Some manipulators also affect input streams. `std::boolalpha` appears in both tables because it changes how bools are both printed and read.

Input manipulators:

On	Off	Description
`std::boolalpha`	`std::noboolalpha`	read `true`/`false` as `bool` instead of `1`/`0`
`std::hex`	`std::dec`	read integers as hexadecimal
`std::oct`	`std::dec`	read integers as octal
`std::skipws`	`std::noskipws`	skip leading whitespace before `>>` (default on)
`std::ws`	—	consume all whitespace right now (one-shot)

`std::hex` and `std::oct` also work on output — `std::cout << std::hex << 255` prints `ff`.

```cpp
#include <iomanip>
#include <iostream>

int main()
{
    bool on_tour = true;
    std::cout << on_tour << std::endl;                 // 1
    std::cout << std::boolalpha << on_tour << std::endl; // true

    double score = 9.87654;
    std::cout << std::fixed << std::setprecision(2);
    std::cout << std::setw(10) << score << std::endl;  //       9.88

    return 0;
}
```

Tip: Prefer `std::format` (Chapter 10) for new code. Stream manipulators are **sticky** — once set, they stay in effect for every subsequent output operation on that stream, which can cause surprising formatting changes later in your program.

Wut: `std::setw` is the exception to the sticky rule — it resets after each `<<` operation. Every other manipulator stays in effect until you explicitly change it.

String Streams

Sometimes you want to build a string piece by piece, or parse values out of a string. String streams let you treat a `std::string` like a stream. They live in the `<sstream>` header.

There are three flavors:

- `std::ostringstream` — output only (writing into a string)
- `std::istringstream` — input only (reading from a string)
- `std::stringstream` — both input and output

Their constructors:

```
std::ostringstream();                    // default
std::istringstream(const std::string& s); // from string
```

`std::stringstream` has both constructors.

Building Strings with std::ostringstream

An `std::ostringstream` lets you use `<<` to build a string the same way you use `std::cout` to print to the screen.

```
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    std::ostringstream oss;

    oss << "Man, it's a hot one" << " — " << 1999;
    std::string result = oss.str();

    std::cout << result << std::endl;

    return 0;
}
```

Output:

```
Man, it's a hot one — 1999
```

You stream data into `oss` just like you would into `std::cout`. When you are done, call `.str()` to get the built string:

```
std::string str() const;         // get the string
void str(const std::string& s);  // replace the string
```

This is useful when you need to construct a string from mixed types — integers, floats, other strings — without worrying about manual conversion.

Parsing Strings with std::istringstream

An `std::istringstream` lets you use `>>` to read values out of a string, just like reading from `std::cin`.

```
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    std::string data = "I get knocked down 7 times";
    std::istringstream iss(data);

    std::string word;
```

```
    while (iss >> word) {
        std::cout << "[" << word << "]" << std::endl;
    }

    return 0;
}
```

Output:

```
[I]
[get]
[knocked]
[down]
[7]
[times]
```

The `>>` operator reads one whitespace-delimited token at a time, just like `std::cin >>`. When there is nothing left to read, the stream evaluates to `false` and the loop ends.

You can also extract typed values:

```
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    std::string data = "42 3.14 hola";
    std::istringstream iss(data);

    int n;
    double d;
    std::string s;

    iss >> n >> d >> s;
    std::cout << "int: " << n << ", double: " << d
              << ", string: " << s << std::endl;

    return 0;
}
```

Output:

```
int: 42, double: 3.14, string: hola
```

Tip: String streams are great for converting between strings and numbers. To turn an `int` into a `std::string`, stream it into an `std::ostringstream` and call `.str()`. To turn a `std::string` into an `int`, put it in an `std::istringstream` and extract with `>>`.

File Streams

File streams let you read from and write to files on disk. They live in the `<fstream>` header and work exactly like `std::cin` and `std::cout`, but connected to files instead of the keyboard and screen.

- `std::ifstream` — input file stream (reading)
- `std::ofstream` — output file stream (writing)
- `std::fstream` — both reading and writing

Writing to a File

`std::ofstream` opens a file for writing. Its constructor takes the filename:

```
std::ofstream(const std::string& filename);
```

```
#include <fstream>
#include <iostream>

int main()
{
    std::ofstream outfile("setlist.txt");

    if (!outfile) {
        // std::cerr is the standard error stream --- like std::cout, but
        // intended for error messages.  It is unbuffered, so messages
        // appear immediately.
        std::cerr << "Could not open file for writing" << std::endl;
        return 1;
    }

    outfile << "Closing Time" << std::endl;
    outfile << "Smooth" << std::endl;
    outfile << "Tubthumping" << std::endl;

    outfile.close();
    std::cout << "Setlist saved!" << std::endl;

    return 0;
}
```

You create an `std::ofstream` by passing the filename to its constructor. Then you use `<<` exactly like you would with `std::cout`. When you are done, call `.close()`:

```
void close();
```

Tip: Always check if a file opened successfully before using it. If the file could not be opened, the stream evaluates to `false`. Using a stream that failed to open will silently do nothing — no errors, no data, just silence.

Reading from a File

To read a file, use `std::ifstream`. Its constructor mirrors `std::ofstream`:

```
std::ifstream(const std::string& filename);
```

The most common pattern is reading line by line with `std::getline`:

```
#include <fstream>
#include <iostream>
#include <string>

int main()
{
    std::ifstream infile("setlist.txt");

    if (!infile) {
```

```
        std::cerr << "Could not open setlist.txt" << std::endl;
        return 1;
    }

    std::string line;
    int count = 0;

    while (std::getline(infile, line)) {
        ++count;
        std::cout << count << ": " << line << std::endl;
    }

    infile.close();

    return 0;
}
```

Output (assuming the file from the previous example exists):

```
1: Closing Time
2: Smooth
3: Tubthumping
```

`std::getline(infile, line)` reads one full line from the file into `line`. When the end of the file is reached, `std::getline` returns a value that evaluates to `false`, ending the loop.

You can also read word by word using `>>`:

```
std::ifstream infile("setlist.txt");
std::string word;

while (infile >> word) {
    std::cout << word << std::endl;
}
```

This would print each word on its own line, splitting on whitespace.

Trap: Do not forget to check if the file opened before reading. A common beginner mistake is to skip the check and then wonder why the program produces no output. The stream just silently fails.

Closing Files

You should call `.close()` when you are done with a file. However, file streams automatically close when they go out of scope (when the variable is destroyed at the end of a block).

```
void write_log()
{
    std::ofstream log("event.log");
    log << "It's closing time" << std::endl;
    // log.close() happens automatically here
}
```

Tip: While files close automatically when the stream goes out of scope, calling `.close()` explicitly makes your intent clear and ensures data is flushed immediately.

File Modes

By default, `std::ofstream` truncates the file — it erases any existing content when it opens. You can change this behavior with **file mode flags** passed as a second argument to the constructor:

```
std::ofstream(const std::string& filename, std::ios::openmode mode);
```

Flag	Meaning
`std::ios::out`	open for writing (default for `ofstream`)
`std::ios::app`	append to the end of the file
`std::ios::in`	open for reading (default for `ifstream`)
`std::ios::binary`	open in binary mode (no text translations)
`std::ios::trunc`	truncate file on open (default with `out`)

You combine multiple flags with the `|` operator — the same bitwise OR you learned in Chapter 4:

```
#include <fstream>
#include <iostream>

int main()
{
    // Append to a log file instead of overwriting it
    std::ofstream log("setlist.log", std::ios::out | std::ios::app);

    if (!log) {
        std::cerr << "Could not open log" << std::endl;
        return 1;
    }

    log << "Closing Time" << std::endl;
    log.close();

    return 0;
}
```

Each time you run this program, it adds a line to `setlist.log` instead of replacing the file.

Tip: The `|` operator here is the same bitwise OR from Chapter 4. File mode flags are implemented as bitmasks — each flag sets a different bit, and combining them with `|` turns on multiple bits at once.

Putting It All Together

Here is a program that writes data to a file, reads it back, and uses string streams to parse each line:

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    std::ofstream outfile("setlist.txt");
    if (!outfile) {
```

```
        std::cerr << "Could not open file" << std::endl;
        return 1;
    }

    outfile << "ClosingTime 1998" << std::endl;
    outfile << "Tubthumping 1997" << std::endl;
    outfile << "Smooth 1999" << std::endl;
    outfile.close();

    std::ifstream infile("setlist.txt");
    if (!infile) {
        std::cerr << "Could not open setlist.txt" << std::endl;
        return 1;
    }

    std::string line;
    std::ostringstream summary;
    int count = 0;

    while (std::getline(infile, line)) {
        std::istringstream iss(line);
        std::string song;
        int year;
        iss >> song >> year;
        ++count;
        summary << count << ". " << song << " (" << year << ")" << std::endl;
    }

    infile.close();

    std::cout << "Setlist:" << std::endl;
    std::cout << summary.str();

    return 0;
}
```

Output:

```
Setlist:
1. ClosingTime (1998)
2. Tubthumping (1997)
3. Smooth (1999)
```

Key Points

- All C++ streams share the same `<<` and `>>` interface — once you learn one, you know them all.
- `std::ostringstream` builds strings from mixed types; `std::istringstream` parses values out of strings.
- `std::ofstream` writes to files; `std::ifstream` reads from files.
- Always check if a file stream opened successfully before using it.
- File streams close automatically when they go out of scope, but explicit `.close()` makes intent clear.
- Stream manipulators (`std::setw`, `std::setprecision`, `std::fixed`, `std::boolalpha`) control formatting but are largely superseded by `std::format` (Chapter 10).
- File mode flags (`std::ios::app`, `std::ios::binary`, etc.) control how files are opened; combine them with `|`.

Exercises

1. What does the following program print?

```cpp
#include <sstream>
#include <iostream>

int main()
{
    std::ostringstream oss;
    oss << 10 << " + " << 20 << " = " << 10 + 20;
    std::cout << oss.str() << std::endl;
    return 0;
}
```

2. What does this program print?

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

int main()
{
    std::istringstream iss("100 hola 3.14");
    int n;
    std::string s;
    double d;

    iss >> n >> s >> d;
    std::cout << d << " " << n << " " << s << std::endl;
    return 0;
}
```

3. What is wrong with this code?

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

int main()
{
    std::ifstream infile("data.txt");
    std::string line;

    while (std::getline(infile, line)) {
        std::cout << line << std::endl;
    }

    return 0;
}
```

4. What is wrong with this file-writing code?

```cpp
#include <fstream>

int main()
{
    std::ofstream out;
```

```
    out << "Yo me la paso bien" << std::endl;
    out.close();
    return 0;
}
```

5. Why is it useful that string streams, file streams, and `std::cout`/`std::cin` all share the same `<<` and `>>` interface?

6. Write a program that reads three song names from the user using `std::getline`, builds a single string containing all three songs separated by `/` using an `std::ostringstream`, writes that string to a file called `favorites.txt`, then reads the file back and prints its contents.

7. What does this program print?

```
#include <iomanip>
#include <iostream>

int main()
{
    std::cout << std::boolalpha << (5 > 3) << std::endl;
    std::cout << std::fixed << std::setprecision(1);
    std::cout << 3.14159 << std::endl;
    return 0;
}
```

8. What happens if you open an `std::ofstream` with `std::ios::app` and write to it? How does this differ from the default behavior?

9. Given the input string `"Closing Time 1998 Smooth 1999"`, how many times will this loop iterate?

```
std::istringstream iss("Closing Time 1998 Smooth 1999");
std::string word;
int count = 0;
while (iss >> word) {
    count++;
}
```

   What is the final value of `count`?

10. **What does this print?**

```
#include <sstream>
#include <iostream>
#include <string>

int main()
{
    std::stringstream ss;
    ss << "year " << 1999;
    std::string word;
    int year{};
    ss >> word >> year;
    std::cout << "[" << word << "] [" << year << "]\n";
    return 0;
}
```

   `std::stringstream` is bidirectional — you can write into it with `<<` and then read out of it with `>>`. Walk through what is in the stream after the `<<` line and what each `>>` extracts.

11. **Calculation:** What is the value of each of these `std::ios_base::openmode` expressions, and what does each combination do?

    ```
    std::ios::out
    std::ios::out | std::ios::app
    std::ios::out | std::ios::trunc
    std::ios::in  | std::ios::out | std::ios::binary
    ```

    Why do you OR the flags together with `|` instead of using `+` or `,`?

12. **Write a program** that opens a file called `oldies.txt`, reads each line into a `std::vector<std::string>`, and prints them. If the file cannot be opened, write a clear error message to `std::cerr` (not `std::cout`) and return a non-zero exit code. Why does sending the error to `std::cerr` matter even though both streams print to the same terminal by default?

10. std::format and std::print

In Chapter 9, you learned how to read and write data using streams. Formatting that output — aligning columns, controlling decimal places, padding with characters — used to require verbose stream manipulators. C++20 introduced `std::format`, and C++23 added `std::print` and `std::println`, giving you clean, readable formatting in one step. In this chapter, you will learn how to use these modern formatting tools.

std::format

Before C++20, formatting output with `std::cout` could be awkward. Mixing text and values with lots of `<<` operators gets hard to read quickly.

C++20 introduced `std::format` in the `<format>` header. It uses format strings with `{}` placeholders, similar to Python's f-strings or C's `printf`.

```
std::string format(format_string fmt, Args... args);
```

`std::format` takes a format string and one or more arguments, and returns a `std::string`.

```
#include <format>
#include <iostream>
#include <string>

int main()
{
    std::string artist = "Santana";
    int year = 1999;

    std::string msg = std::format("{} — Smooth ({})", artist, year);
    std::cout << msg << std::endl;

    return 0;
}
```

Output:

```
Santana — Smooth (1999)
```

`std::format` returns a `std::string`. Each `{}` is replaced by the next argument, in order. This is called **implicit** argument numbering.

Implicit vs. Indexed Arguments

You can also use **indexed** arguments by putting a number inside the braces. The number refers to the zero-based position of the argument:

```
std::string msg = std::format("{1} — {0} ({2})", "Santana", "Smooth", 1999);
// "Smooth — Santana (1999)"
```

Indexed arguments let you reorder or reuse arguments without changing the argument list.

```
std::format("{0}, {0}, {0}!", "yeah");  // "yeah, yeah, yeah!"
```

Trap: You cannot mix implicit `{}` and indexed `{0}` in the same format string. `std::format("{},{1}", 1, "hi")` is an error. Pick one style and use it consistently within each format string.

Format Specifiers

You can control how values are formatted by adding specifiers inside the braces.

Width and alignment:

```
// Right-align in a field of 10 characters
std::format("{:>10}", "hola");    // "      hola"

// Left-align in a field of 10 characters
std::format("{:<10}", "hola");    // "hola      "

// Center in a field of 10 characters
std::format("{:^10}", "hola");    // "   hola   "
```

Fill characters:

```
std::format("{:*>10}", "hola");   // "******hola"
std::format("{:-^20}", "Smooth"); // "-------Smooth-------"
```

Sign:

The sign specifier controls how positive numbers are displayed:

```
std::format("{:+}", 42);          // "+42" (always show sign)
std::format("{:-}", 42);          // "42"  (negatives only)
std::format("{: }", 42);          // " 42" (space where + would go)
```

Alternate form (`#`) and zero-padding (`0`):

The `#` flag shows a prefix that identifies the number base. The `0` flag pads with zeros instead of spaces:

```
std::format("{:#x}", 255);        // "0xff"  (hex with 0x prefix)
std::format("{:#b}", 10);         // "0b1010" (binary with 0b prefix)
std::format("{:05}", 42);         // "00042" (zero-padded to width 5)
std::format("{:#010x}", 255);     // "0x000000ff" (combined)
```

Number formatting:

```
std::format("{:d}", 42);          // "42" (decimal integer)
std::format("{:x}", 255);         // "ff" (hexadecimal)
std::format("{:o}", 8);           // "10" (octal)
std::format("{:b}", 10);          // "1010" (binary)
```

Floating-point precision:

```
std::format("{:.2f}", 3.14159);   // "3.14"
std::format("{:.4f}", 2.5);       // "2.5000"
std::format("{:10.2f}", 3.14);    // "      3.14"
```

Here is a more complete example:

```
#include <format>
#include <iostream>

int main()
{
    std::cout << std::format("{:<20} {:>5} {:>8}", "Song", "Year", "Score")
              << std::endl;
    std::cout << std::format("{:<20} {:>5} {:>8.1f}",
        "Wonderwall", 1995, 9.5) << std::endl;
    std::cout << std::format("{:<20} {:>5} {:>8.1f}", "Jumper", 1997, 9.8)
```

```
              << std::endl;
    std::cout << std::format("{:<20} {:>5} {:>8.1f}",
        "Say My Name", 1999, 8.7) << std::endl;

    return 0;
}
```

Output:

```
Song                 Year    Score
Wonderwall           1995      9.5
Jumper               1997      9.8
Say My Name          1999      8.7
```

Tip: `std::format` is much easier to read than chaining `<<` operators with `std::setw` and `std::setprecision`. If your compiler supports C++20 or later, prefer `std::format` for any non-trivial formatting.

std::print and std::println

C++23 took things one step further with `std::print` and `std::println` in the `<print>` header. These combine `std::format` and `std::cout` into a single call.

```
void print(format_string fmt, Args... args);
void println(format_string fmt, Args... args);
```

```
#include <print>

int main()
{
    std::println("You get what you give, don't let go");
    std::print("Track {:d}: {}", 1, "You Get What You Give");
    std::println("");

    double score = 9.5;
    std::println("Rating: {:.1f}/10", score);

    return 0;
}
```

Output:

```
You get what you give, don't let go
Track 1: You Get What You Give
Rating: 9.5/10
```

`std::println` prints a formatted string followed by a newline. `std::print` prints without a trailing newline.

These are the modern replacements for `std::cout <<`. They are shorter to write, easier to read, and handle formatting in one step.

Wut: `std::print` and `std::println` require C++23 support. Not all compilers support them yet. If your compiler does not have `<print>`, you can use `std::format` with `std::cout` to achieve the same result.

Putting It All Together

Here is a program that uses the file I/O from Chapter 9 along with the formatting tools from this chapter:

```
#include <fstream>
#include <format>
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    std::ofstream outfile("puntuaciones.txt");
    if (!outfile) {
        std::cerr << "No puedo abrir el archivo" << std::endl;
        return 1;
    }

    outfile << "Wonderwall 9.5" << std::endl;
    outfile << "Jumper 9.8" << std::endl;
    outfile << "SayMyName 8.7" << std::endl;
    outfile.close();

    std::ifstream infile("puntuaciones.txt");
    if (!infile) {
        std::cerr << "Could not open file" << std::endl;
        return 1;
    }

    std::string line;
    std::cout << std::format("{:<15} {:>6}", "Song", "Score")
              << std::endl;
    std::cout << std::string(22, '-') << std::endl;

    while (std::getline(infile, line)) {
        std::istringstream iss(line);
        std::string song;
        double score;

        iss >> song >> score;
        std::cout << std::format("{:<15} {:>6.1f}", song, score)
                  << std::endl;
    }

    infile.close();

    return 0;
}
```

Output:

```
Song             Score
----------------------
Wonderwall           9.5
Jumper               9.8
SayMyName            8.7
```

Key Points

- `std::format` (C++20) uses `{}` placeholders to produce formatted strings.
- Use `{0}`, `{1}`, etc., to reorder or reuse arguments. You cannot mix implicit `{}` and indexed `{0}` in the same format string.
- `std::print` and `std::println` (C++23) combine formatting and output in one step.
- Format specifiers control width, alignment, precision, and base (e.g., `{:>10.2f}`, `{:x}`).
- Fill characters, alignment (`<`, `>`, `^`), and number bases (`d`, `x`, `o`, `b`) give you fine-grained control.
- `std::format` returns a `std::string`; `std::print` and `std::println` write directly to the output.

Exercises

1. What does `std::format("{:>8.2f}", 3.1)` produce? How many characters wide is the result?
2. Why might you prefer `std::format` over chaining `<<` operators with `std::cout`? Give at least two reasons.
3. What is the difference between `std::print` and `std::println`?
4. What does `std::format("{:*^20}", "Hola")` produce?
5. What is wrong with this code?

   ```
   std::string result = std::format("{} scored {1} points", name, score);
   ```

6. Write a program that asks the user for three song names and three scores (as doubles), writes them to a file called `rankings.txt` (one song and score per line), then reads the file back and prints a formatted table with columns for song name and score, right-aligning the scores to one decimal place.
7. **What does this print?**

   ```
   std::println("{1} - {0} ({2})", "Backstreet Boys", "I Want It That Way", 1999);
   ```

   Now change every `{0}`/`{1}`/`{2}` to plain `{}` and predict the output. What is the rule about mixing indexed and implicit placeholders in the same format string?
8. **Calculation:** What does each of these `std::format` calls produce?

   ```
   std::format("{:+d}", 42)
   std::format("{:+d}", -42)
   std::format("{: d}", 42)
   std::format("{:05d}", 42)
   std::format("{:+06d}", -42)
   ```

   For each one, write down the exact characters in the resulting string, including any spaces or zeros.
9. **What does this print?**

   ```
   int n = 255;
   std::println("{:#x}", n);
   std::println("{:#o}", n);
   std::println("{:#b}", n);
   std::println("{:08b}", n);
   ```

   What does the `#` flag do, and what does the `08` in the last line do?
10. **What does this print?**

    ```
    std::string title = "Smells Like Teen Spirit";
    std::println("[{:.5}]", title);
    std::println("[{:<10.5}]", title);
    std::println("[{:>10.5}]", title);
    ```

For string arguments, what does the precision (`.5`) mean? How is that different from precision on a `double`?

11. **Write a program** that takes three integers, formats them into a single `std::string` using `std::format`, and prints each integer in three different ways:

    - decimal in a 6-character field, right-aligned
    - hexadecimal with the `0x` prefix and zero-padded to 8 hex digits
    - binary with the `0b` prefix, zero-padded to 16 bits

    Use a single `std::format` call per row so you practice combining width, fill, and base specifiers in the same format string.

11. Exceptions

So far, when something goes wrong in your programs you have printed an error message and returned early. That works when the error happens in `main()`, but what about a function buried three or four calls deep? You would have to thread error codes back through every function in the chain, and every caller would have to check the return value — tedious and easy to get wrong. C++ provides **exceptions**, a mechanism that lets a function signal an error and lets code much higher up in the call stack handle it. C++23 also introduces `std::expected`, which gives you a way to return either a value or an error without the overhead of exceptions. In this chapter you will learn how to throw and catch exceptions, how the stack unwinds when an exception is thrown, when to use `noexcept`, and how `std::expected` offers a lightweight alternative.

Throwing Exceptions

When a function encounters a situation it cannot handle, it **throws** an exception using the `throw` keyword:

```
#include <stdexcept>
#include <string>

int parse_track(const std::string &s) {
    int n = std::stoi(s);
    if (n < 1) {
        throw std::out_of_range("track number must be positive");
    }
    return n;
}
```

When `throw` executes, the function stops immediately. It does not return a value — control leaves the function and travels up the call stack looking for something that can handle the error.

The `<stdexcept>` header provides several standard exception types, all derived from `std::exception`:

Type	When to use
`std::runtime_error`	general errors detected at runtime
`std::out_of_range`	a value is outside an acceptable range
`std::invalid_argument`	an argument does not make sense
`std::logic_error`	a bug in the program's logic
`std::overflow_error`	arithmetic overflow

All of them take a `std::string` message in their constructor and provide a `what()` member function that returns it.

Catching Exceptions

To handle an exception, wrap the code that might throw in a **try block** and follow it with one or more **catch blocks**:

```
#include <iostream>
#include <stdexcept>
#include <string>

int parse_track(const std::string &s) {
    int n = std::stoi(s);
    if (n < 1) {
        throw std::out_of_range("track number must be positive");
```

```
    }
    return n;
}

int main()
{
    try {
        int track = parse_track("0");
        std::cout << "Track: " << track << std::endl;
    } catch (const std::out_of_range &e) {
        std::cout << "Error: " << e.what() << std::endl;
    }

    return 0;
}
```

Output:

```
Error: track number must be positive
```

When `parse_track` throws `std::out_of_range`, the rest of the `try` block is skipped and the matching `catch` block runs. After the `catch` block finishes, execution continues normally after it — the program does not crash.

Multiple Catch Blocks

You can have multiple `catch` blocks to handle different exception types. The compiler tries them in order and uses the first one that matches:

```
#include <iostream>
#include <stdexcept>
#include <string>

int parse_volume(const std::string &s) {
    int v = std::stoi(s);
    if (v < 0 || v > 11) {
        throw std::out_of_range("volume must be 0-11");
    }
    return v;
}

int main()
{
    try {
        int v = parse_volume("abc");
        std::cout << "Volume: " << v << std::endl;
    } catch (const std::out_of_range &e) {
        std::cout << "Out of range: " << e.what() << std::endl;
    } catch (const std::invalid_argument &e) {
        std::cout << "Bad input: " << e.what() << std::endl;
    }

    return 0;
}
```

Output:

```
Bad input: stoi
```

Here `std::stoi("abc")` throws `std::invalid_argument`, so the second `catch` block handles it. If you passed `"99"` instead, `parse_volume` would throw `std::out_of_range` and the first `catch` would handle it.

Catching Everything

You can catch any exception with `catch (...)`:

```
try {
    risky_function();
} catch (const std::exception &e) {
    std::cout << "Known error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "Unknown error" << std::endl;
}
```

The `catch (...)` block is a last resort — it catches any exception, including types that do not derive from `std::exception`. Always catch specific types first and use `catch (...)` only as a safety net.

Tip: Always catch exceptions by `const` reference (`const std::exception &e`). Catching by value makes a copy and can slice off information from derived exception types.

Stack Unwinding

When an exception is thrown, C++ **unwinds the stack** — it walks back up the call chain, destroying local variables in each function along the way, until it finds a matching `catch` block. This means destructors run automatically, which is critical for cleaning up resources.

```
#include <iostream>
#include <stdexcept>
#include <string>

struct Song {
    std::string title;
    Song(const std::string &t) : title(t) {
        std::cout << "  created: " << title << std::endl;
    }
    ~Song() {
        std::cout << "  destroyed: " << title << std::endl;
    }
};

void deep_function() {
    Song s("The Freshmen");
    throw std::runtime_error("something went wrong");
}

void middle_function() {
    Song s("Save Tonight");
    deep_function();
}

int main()
```

```
{
    try {
        middle_function();
    } catch (const std::runtime_error &e) {
        std::cout << "Caught: " << e.what() << std::endl;
    }

    return 0;
}
```

Output:

```
  created: Save Tonight
  created: The Freshmen
  destroyed: The Freshmen
  destroyed: Save Tonight
Caught: something went wrong
```

Even though no function returned normally, both `Song` destructors ran. `deep_function`'s `Song` ("The Freshmen") is destroyed first (most recent), then `middle_function`'s ("Save Tonight"). This automatic cleanup during stack unwinding is why destructors are so important — and why you should manage resources through objects rather than raw `new`/`delete`.

Trap: If a destructor throws an exception during stack unwinding (while another exception is already in flight), the program calls `std::terminate()` and crashes. Never throw from a destructor.

noexcept

The `noexcept` keyword promises the compiler that a function will not throw any exceptions:

```
int add(int a, int b) noexcept {
    return a + b;
}
```

If a `noexcept` function does throw (for example, by calling a function that throws), the program calls `std::terminate()` immediately — there is no stack unwinding, no catch blocks, just a crash.

`noexcept` is not just documentation — the compiler uses it to generate more efficient code. Standard library containers like `std::vector` check whether your move operations are `noexcept` before deciding whether to move or copy during reallocation. If your move constructor is `noexcept`, the vector moves elements (fast). If it is not, the vector falls back to copying (slow) because a failed move would leave the container in a broken state.

Tip: Mark functions `noexcept` when you are certain they will not throw. This is especially important for move constructors, move assignment operators, and destructors.

Trap: The compiler does not verify that a `noexcept` function actually avoids throwing. If you mark a function `noexcept` but it calls something that throws, you get `std::terminate()` at runtime with no warning at compile time.

std::expected

Exceptions are powerful, but they are not always the right tool. They are best for truly exceptional situations — file not found, out of memory, network failure. For errors that are a normal part of a function's contract (like parsing invalid user input), the overhead of exception handling can be unnecessary.

C++23 introduces `std::expected<T, E>` in the `<expected>` header. It holds either a value of type `T` (the success case) or an error of type `E` (the failure case) — but never both.

```
std::expected<T, E>
```

You return the value normally for success, and wrap the error in `std::unexpected` for failure:

```
#include <expected>
#include <iostream>
#include <string>

std::expected<int, std::string> parse_track(const std::string &s) {
    try {
        int n = std::stoi(s);
        if (n < 1) {
            return std::unexpected("track must be positive");
        }
        return n;
    } catch (...) {
        return std::unexpected("not a number");
    }
}

int main()
{
    auto result = parse_track("5");
    if (result) {
        std::cout << "Track: " << *result << std::endl;
    }

    auto error = parse_track("abc");
    if (!error) {
        std::cout << "Error: " << error.error() << std::endl;
    }

    return 0;
}
```

Output:

```
Track: 5
Error: not a number
```

Use `*result` or `result.value()` to get the value, and `result.error()` to get the error. The boolean check (`if (result)`) tells you whether it holds a value or an error.

Exceptions vs std::expected

When should you use which?

	Exceptions	`std::expected`
Best for	rare, truly exceptional failures	expected, routine failures
Error path	unwinds the stack	returns normally
Caller must check?	no — propagates automatically	yes — must inspect the return value
Performance	zero cost when no exception is thrown; expensive when thrown	small constant cost (size of the return type)

A good rule of thumb: if the caller is *likely* to handle the error immediately, use `std::expected`. If the error should propagate up several layers, use exceptions.

Tip: `std::expected` makes error handling explicit and visible in the return type. This is especially useful for functions where failure is a normal outcome, like parsing user input or looking up a key in a map.

Key Points

- Use `throw` to signal an error and `try/catch` to handle it.
- The standard exception types in `<stdexcept>` cover most common error categories.
- Always catch exceptions by `const` reference.
- Stack unwinding destroys local variables automatically when an exception propagates — this is why resource management through objects (RAII) matters.
- Never throw from a destructor.
- `noexcept` promises a function will not throw; violating the promise calls `std::terminate()`.
- Mark move constructors, move assignment operators, and destructors `noexcept`.
- `std::expected<T, E>` (C++23) returns either a value or an error without using exceptions.
- Use exceptions for rare failures that should propagate; use `std::expected` for routine errors the caller handles immediately.

Exercises

1. What does the following program print?

```cpp
#include <iostream>
#include <stdexcept>

void step3() { throw std::runtime_error("oops"); }
void step2() { step3(); }
void step1() { step2(); }

int main()
{
    try {
        step1();
        std::cout << "A" << std::endl;
    } catch (const std::runtime_error &e) {
        std::cout << "B: " << e.what() << std::endl;
    }
    std::cout << "C" << std::endl;
```

```
    return 0;
}
```

2. What is wrong with this code?

```
try {
    int n = std::stoi(input);
} catch (const std::out_of_range &e) {
    std::cout << "out of range" << std::endl;
} catch (const std::exception &e) {
    std::cout << "error" << std::endl;
} catch (const std::invalid_argument &e) {
    std::cout << "bad input" << std::endl;
}
```

3. Why should you always catch exceptions by `const` reference rather than by value?

4. What does the following program print?

```
#include <iostream>
#include <stdexcept>
#include <string>

struct Amp {
    std::string name;
    Amp(const std::string &n) : name(n) {
        std::cout << name << " on" << std::endl;
    }
    ~Amp() {
        std::cout << name << " off" << std::endl;
    }
};

void soundcheck() {
    Amp a("Marshall");
    Amp b("Fender");
    throw std::runtime_error("feedback!");
}

int main()
{
    try {
        soundcheck();
    } catch (...) {
        std::cout << "handled" << std::endl;
    }
    return 0;
}
```

5. Will this code compile? If so, what happens when `play()` is called?

```
void load(const std::string &file) {
    throw std::runtime_error("file not found");
}

void play() noexcept {
    load("track01.wav");
```

```
}
```

6. What is the output of this program?

```
#include <expected>
#include <iostream>
#include <string>

std::expected<int, std::string> divide(int a, int b) {
    if (b == 0) {
        return std::unexpected("division by zero");
    }
    return a / b;
}

int main()
{
    auto r1 = divide(10, 3);
    auto r2 = divide(10, 0);

    if (r1) std::cout << *r1 << std::endl;
    if (!r2) std::cout << r2.error() << std::endl;

    return 0;
}
```

7. When would you use `std::expected` instead of throwing an exception? Give an example scenario for each.

8. How many destructors run before the `catch` block executes?

```
struct Song {
    std::string title;
    Song(const std::string &t) : title(t) {}
    ~Song() { std::cout << "destroyed: " << title << std::endl; }
};

void inner() {
    Song a("Torn");
    Song b("Vogue");
    throw std::runtime_error("oops");
}

void outer() {
    Song c("Iris");
    inner();
}

int main() {
    try {
        outer();
    } catch (...) {
        std::cout << "caught" << std::endl;
    }
    return 0;
}
```

9. Write a function `safe_sqrt` that takes a `double` and returns `std::expected<double, std::string>`. If the input is negative, return an error message. Otherwise, return the square root. Test it in `main()` with both a positive and a negative value.

10. **Where is the bug?**

```cpp
#include <iostream>
#include <stdexcept>

int main()
{
    try {
        throw std::out_of_range("nope");
    }
    catch (...) {
        std::cout << "anything\n";
    }
    catch (const std::out_of_range &e) {
        std::cout << "out_of_range: " << e.what() << "\n";
    }
    return 0;
}
```

Catch handlers are tried in source order, top to bottom. Why is the second `catch` block effectively dead code? How would you reorder the handlers so that `out_of_range` is caught specifically and `catch(...)` only acts as a final safety net?

11. **Write a program** that defines a function

```cpp
int parse_age(const std::string &s);
```

that converts `s` to an integer using `std::stoi` and then returns it. Throw `std::invalid_argument("not a number")` if `std::stoi` itself throws `std::invalid_argument`, and throw `std::out_of_range("age must be 0..150")` if the parsed number is outside the range `[0, 150]`. In `main`, call `parse_age` on three inputs — `"42"`, `"abc"`, and `"-1"` — inside `try`/`catch` blocks that catch each of the two exception types separately and print a different message for each one.

12. Classes

In Chapter 2 you learned that structures group related data together under one name. Since then you have been using structs to bundle fields like `title`, `artist`, and `year` into a single variable. But structs have a problem: all their members are public by default, so any code can reach in and set `year` to -5 or `title` to an empty string. There is no way to enforce rules about valid values, and the functions that operate on a struct live separately from the data they work on. Classes solve this by bundling data *and* behavior into a single unit with control over who can access what. This is the foundation of **object-oriented programming** in C++.

In Chapter 2 you learned that C++ calls any region of memory with a type an **object**. A class is a blueprint — it describes what data and behavior a type has. When you create a variable of that class type, the object you get is called an **instance** of the class. In everyday C++ conversation, "object" and "instance" are used interchangeably when talking about classes. In this chapter you will learn how to define classes, control access to their members, write constructors and destructors, and teach the compiler what `<<` and `==` mean for your own types.

From Structs to Classes

You already know that a `struct` groups related variables together. Here is a simple struct from Chapter 2:

```
struct Song {
    std::string title;
    std::string artist;
    int year;
};
```

You can create a `Song` and access its members with the `.` operator:

```
Song s;
s.title = "Enter Sandman";
s.artist = "Metallica";
s.year = 1991;
```

This works, but as mentioned above, nothing prevents invalid data — anyone can set `year` to -5. Let's fix that.

Access Specifiers

Access specifiers control who can see and use the members of a class. There are three:

- **`public`**: accessible from anywhere.
- **`private`**: accessible only from inside the class itself.
- **`protected`**: accessible from inside the class and from derived classes (classes that extend your class — a topic for a more advanced C++ book).

A **class** is almost identical to a struct, but with one key difference: members are **private by default**, meaning code outside the class cannot access them directly. This lets you control how your data is used.

```
class Song {
public:
    std::string title;
    std::string artist;
    int year;
};
```

Wut: In C++, `struct` and `class` are almost the same thing. The only difference is that members of a `struct` are `public` by default, while members of a `class` are `private` by default. You could use either one, but by convention `class` is used when you want to bundle data with behavior.

```cpp
class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    void set_title(const std::string &t) { title = t; }
    std::string get_title() const { return title; }

    void set_year(int y) {
        if (y > 0) {
            year = y;
        }
    }
    int get_year() const { return year; }
};
```

Now `title`, `artist`, and `year` are private. The only way to set the year is through `set_year()`, which checks that the value is positive. This pattern of providing functions to access private data is common in C++.

Tip: Not every private member needs a getter and setter. Only expose what other code actually needs. The whole point of making data private is to keep the interface small and controlled.

Constructors

A **constructor** is a special function that runs automatically when an object is created. Its job is to set up the object so it starts in a valid state.

The constructor has the same name as the class and no return type — not even `void`.

Default Constructor

A **default constructor** takes no arguments:

```cpp
class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    Song() {
        title = "Unknown";
        artist = "Unknown";
        year = 0;
    }
};
```

Now when you create a `Song` without arguments, it starts with sensible defaults:

```cpp
Song s;  // calls the default constructor
// s.title is "Unknown", s.artist is "Unknown", s.year is 0
```

Parameterized Constructor

A **parameterized constructor** takes arguments so you can initialize the object with specific values:

```
class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    Song() : title("Unknown"), artist("Unknown"), year(0) {}

    Song(const std::string &t, const std::string &a, int y)
        : title(t), artist(a), year(y) {}
};
```

Now you can create songs either way:

```
Song a;                                          // default
Song b("Enter Sandman", "Metallica", 1991);      // parameterized
```

Member Initializer Lists

Notice the `: title(t), artist(a), year(y)` syntax after the constructor's parameter list. This is a **member initializer list**, and it is the preferred way to initialize members in C++.

The member initializer list initializes members directly, rather than default-constructing them and then assigning new values in the body. For simple types like `int`, the difference is negligible. For complex types like `std::string`, the initializer list is more efficient because it avoids constructing a temporary default value that is immediately thrown away.

Tip: Always prefer member initializer lists over assignment in the constructor body. It is more efficient and, for some types like `const` members and references, it is the *only* way to initialize them.

The order of initialization is determined by the order members are *declared* in the class, not the order they appear in the initializer list.

Trap: If you list members in your initializer list in a different order than they are declared, the compiler may warn you. Always match the order of your initializer list to the order of your member declarations.

`explicit` Constructors

A constructor that takes a single argument can be used as an **implicit conversion**. This means the compiler will silently call the constructor to convert one type to another without you asking:

```
class Volume {
public:
    int level;
    Volume(int l) : level(l) {}
};

void play(Volume v) {
    std::cout << "Volume: " << v.level << std::endl;
```

```
}

play(11);   // compiles! the compiler converts 11 to Volume(11)
```

That might be convenient, but it can also cause surprises. The `explicit` keyword prevents this — it forces the caller to construct the object explicitly:

```
class Volume {
public:
    int level;
    explicit Volume(int l) : level(l) {}
};

play(11);           // ERROR: no implicit conversion
play(Volume(11));   // OK: explicit construction
play(Volume{11});   // OK: also explicit
```

Trap: Single-argument constructors without `explicit` are a common source of surprise conversions. The compiler can quietly insert conversions you did not intend, leading to bugs that are hard to spot. As a rule of thumb, mark single-argument constructors `explicit` unless you specifically want implicit conversion.

Constructors with two or more required parameters cannot be called implicitly, so `explicit` matters most on single-argument constructors.

Destructors

A **destructor** is the opposite of a constructor — it runs automatically when an object is destroyed (goes out of scope or is deleted). Its job is to clean up any resources the object owns.

The destructor has the same name as the class, prefixed with `~`, and takes no arguments:

```
class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    Song(const std::string &t, const std::string &a, int y)
        : title(t), artist(a), year(y)
    {
        std::cout << title << " created" << std::endl;
    }

    ~Song() {
        std::cout << title << " destroyed" << std::endl;
    }
};

int main()
{
    Song s("Black Hole Sun", "Soundgarden", 1994);
    std::cout << "doing stuff..." << std::endl;
    return 0;
}
```

Output:

```
Black Hole Sun created
doing stuff...
Black Hole Sun destroyed
```

The destructor is called automatically when `s` goes out of scope at the end of `main()`. You do not call it yourself.

For classes that only contain standard library types like `std::string`, the compiler generates a perfectly good destructor for you. You will see in Chapter 13 that destructors become critical when your class manages its own memory, and in Chapter 14 that the compiler can generate special member functions like copy and move constructors automatically.

Member Functions

Functions declared inside a class are called **member functions** (or methods). They have access to all the class's members, including private ones.

```
class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    Song(const std::string &t, const std::string &a, int y)
        : title(t), artist(a), year(y) {}

    void print() const {
        std::cout << title << " by " << artist
                  << " (" << year << ")"
                  << std::endl;
    }

    bool is_90s() const {
        return year >= 1990 && year <= 1999;
    }
};

int main()
{
    Song s("All Star", "Smash Mouth", 1999);
    s.print();

    if (s.is_90s()) {
        std::cout << "Es de los noventa!" << std::endl;
    }

    return 0;
}
```

Output:

```
All Star by Smash Mouth (1999)
Es de los noventa!
```

Notice the `const` after the parameter list in `print()` and `is_90s()`. This tells the compiler that these functions do not modify the object. You should mark every member function that does not change the object as `const`.

Marking member functions `const` matters more than you might think. When you have a `const` reference or a `const` object, the compiler only allows you to call member functions that are marked `const`. This comes up constantly because, as you saw in Chapter 6, passing objects by `const` reference is the standard way to avoid expensive copies.

Consider a function that takes a `Song` by `const` reference:

```
void show(const Song &s)
{
    s.print();   // OK: print() is const
    s.is_90s();  // OK: is_90s() is const
}
```

This compiles because both `print()` and `is_90s()` are marked `const`. The compiler knows they will not modify the object, so calling them through a `const` reference is safe.

Now imagine `print()` was *not* marked `const`:

```
// BAD: forgot const on print()
void print() {
    std::cout << title << " by " << artist
              << " (" << year << ")" << std::endl;
}
```

The `show()` function above would fail to compile with an error like:

```
error: passing 'const Song' as 'this' argument discards qualifiers
```

The compiler is telling you that `print()` *might* modify the object (since it is not `const`), so calling it on a `const Song` is not allowed.

The fix is simple: add `const` after the parameter list. Get into the habit of marking every member function that does not modify the object as `const`. It documents your intent, catches accidental mutations at compile time, and ensures your class works smoothly with `const` references and objects.

Overloading Member Functions

Sometimes you want the same operation to behave differently depending on what information the caller provides. For example, you might want `print()` to work on its own but also accept an optional label. You saw in Chapter 6 that **function overloading** lets you define multiple functions with the same name as long as they have different signatures (recall from Chapter 0 that a signature is the function's name and parameter list). The same technique works with member functions.

Here is a `Song` class with an overloaded `print()`:

```
class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    Song(const std::string &t, const std::string &a, int y)
        : title(t), artist(a), year(y) {}

    void print() const {
        std::cout << title << " by " << artist
```

```
                  << " (" << year << ")"
                  << std::endl;
    }

    void print(const std::string &label) const {
        std::cout << label << ": " << title << " by " << artist
                  << " (" << year << ")" << std::endl;
    }
};

Song s("Virtual Insanity", "Jamiroquai", 1996);
s.print();                    // calls print()
s.print("Now playing");       // calls print(const std::string &)
```

Output:

```
Virtual Insanity by Jamiroquai (1996)
Now playing: Virtual Insanity by Jamiroquai (1996)
```

You have already seen overloading without realizing it — the `Song` class has two constructors (the default and the parameterized), and those are overloaded functions with the same name `Song`.

Overloading works well when the number or types of parameters differ more substantially. Here is a `Playlist` class with overloaded `add()` methods — one that appends to the end and one that inserts at a specific position:

```
#include <iostream>
#include <string>
#include <vector>

class Playlist {
private:
    std::string name;
    std::vector<std::string> songs;

public:
    Playlist(const std::string &n) : name(n) {}

    void add(const std::string &song) {
        songs.push_back(song);
    }

    void add(const std::string &song, int position) {
        if (position >= 0 && position <= static_cast<int>(songs.size())) {
            songs.insert(songs.begin() + position, song);
        }
    }

    void print() const {
        std::cout << name << ":" << std::endl;
        for (size_t i = 0; i < songs.size(); ++i) {
            std::cout << "  " << i + 1 << ". " << songs[i] << std::endl;
        }
    }
};

int main()
{
```

```
    Playlist p("90s Jams");
    p.add("Torn");                  // calls add(const std::string &)
    p.add("Kiss from a Rose");      // calls add(const std::string &)
    p.add("Basket Case", 1);        // calls add(const std::string &, int)
    p.print();
    return 0;
}
```

Output:

```
90s Jams:
  1. Torn
  2. Basket Case
  3. Kiss from a Rose
```

Overloading keeps your interface clean: one verb for one concept, regardless of how many ways you can call it. Overloading is not limited to member functions — you saw it with free functions in Chapter 6.

Trap: The compiler chooses an overload based on implicit conversions, which can be surprising. If you have `void play(bool)` and `void play(const std::string &)`, calling `play("Torn")` calls the `bool` version — because `"Torn"` is a `const char*`, which converts to `bool` (a non-null pointer is `true`) rather than constructing a `std::string`. Use `play(std::string("Torn"))` to get the version you intended.

The `this` Pointer

When a member function refers to another member by name, the compiler knows to look for it in the current object — writing `title` inside a `Song` method means "this object's `title`". But sometimes you need a reference to the current object *itself*: to pass it to another function, return it from a method, or compare it against another instance. In English, when we want to refer to ourselves we say `me`. C++ doesn't use `me` — it uses another keyword.

When you call `s.print()`, the `print()` function needs to know *which* object it is operating on. If you have ten `Song` objects, there is only one copy of the `print()` code — it needs a way to find the right `title`, `artist`, and `year`. C++ solves this by automatically passing a hidden pointer to the object into every member function call. That pointer is called `this`. You can think of it as the C++ equivalent of `me`.

Before C++11, raw **pointers** were used constantly in C++. Modern C++ has largely moved away from raw pointers in favor of references and smart pointers (Chapter 13), which is why we have made it this far without discussing them. We are still not going to cover pointers in depth here, but you need to know two things about `this`:

- `this` is a **pointer** to the current object. Use `this->member` to access a member variable or function through the pointer.
- `*this` **dereferences** the pointer, giving you the actual object itself.

You usually do not need `this` because member names are accessible directly — when you write `title` inside a member function, the compiler understands you mean `this->title`. But there are a few situations where `this` is necessary.

Resolving Name Conflicts

When a parameter has the same name as a member, `this->` tells the compiler which one you mean:

```
class Song {
private:
    std::string title;
    int year;
```

```
public:
    Song(const std::string &title, int year)
        : title(title), year(year) {}

    void set_year(int year) {
        // this->year is the member, year is the parameter
        this->year = year;
    }
};
```

Without `this->`, the compiler would think `year = year` is assigning the parameter to itself.

Tip: Using member initializer lists (as shown in the constructor above) avoids the `this` ambiguity problem for constructors, as shown in the constructor with `title(title)`. For setter functions, `this->` is a clean way to resolve the conflict.

Returning the Current Object

A member function can return `*this` to give the caller back the object itself. This enables **method chaining** — calling multiple member functions in a single statement:

```
class Playlist {
private:
    std::string name;
    std::vector<std::string> songs;

public:
    Playlist(const std::string &n) : name(n) {}

    Playlist &add(const std::string &song) {
        songs.push_back(song);
        return *this;
    }

    void print() const {
        std::cout << name << ":" << std::endl;
        for (size_t i = 0; i < songs.size(); ++i) {
            std::cout << "  " << i + 1 << ". " << songs[i] << std::endl;
        }
    }
};

Playlist p("90s Jams");
p.add("Torn").add("Kiss from a Rose").add("Basket Case");
p.print();
```

Output:

```
90s Jams:
  1. Torn
  2. Kiss from a Rose
  3. Basket Case
```

Each call to `add()` returns `*this` — the `Playlist` object itself — so the next `.add()` call operates on the same object. Notice the return type is `Playlist &` (a reference), not `Playlist`, so the object is not copied each time.

Copying the Current Object

You can also use `*this` to make a copy of the current object. This is useful when a member function needs to return a modified version without changing the original:

```
class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    Song(const std::string &t, const std::string &a, int y)
        : title(t), artist(a), year(y) {}

    Song with_year(int y) const {
        Song copy = *this;  // copy the current object
        copy.year = y;       // modify only the copy
        return copy;
    }

    void print() const {
        std::cout << title << " by " << artist
                  << " (" << year << ")"
                  << std::endl;
    }
};
```

```
Song original("Torn", "Natalie Imbruglia", 1997);
Song remaster = original.with_year(2023);
original.print();
remaster.print();
```

Output:

```
Torn by Natalie Imbruglia (1997)
Torn by Natalie Imbruglia (2023)
```

`Song copy = *this` creates a full copy of the current `Song`. The function modifies the copy and returns it, leaving the original unchanged. You will see this same pattern in the Operator Overloading section later in this chapter, where `operator+` uses `*this` to make a copy before adding to it.

Default Parameters in Member Functions

Default parameters (introduced in Chapter 6) work in member functions too. The same rules apply: defaults must go at the end of the parameter list, and mixing overloading with defaults can create ambiguity.

Here is the `Playlist` class rewritten to use a default parameter instead of two overloaded `add()` methods:

```
class Playlist {
private:
    std::string name;
    std::vector<std::string> songs;

public:
    Playlist(const std::string &n) : name(n) {}

    void add(const std::string &song, int position = -1) {
```

```
        if (position < 0 || position >= static_cast<int>(songs.size())) {
            songs.push_back(song);
        } else {
            songs.insert(songs.begin() + position, song);
        }
    }

    void print() const {
        std::cout << name << ":" << std::endl;
        for (size_t i = 0; i < songs.size(); ++i) {
            std::cout << "  " << i + 1 << ". " << songs[i] << std::endl;
        }
    }
};
```

Now `add("Torn")` appends to the end (using the default `-1`), and `add("Basket Case", 1)` inserts at position 1.

Tip: When you separate a class into a header and source file, put default parameter values in the *declaration* (the header), not in the *definition* (the source file). The compiler needs to see the defaults at the call site.

Separating Declaration from Definition

As your classes grow, putting everything in one file becomes unwieldy. C++ lets you split a class into a **header file** (`.h`) for the declaration and a **source file** (`.cpp`) for the definitions.

song.h — the declaration:

```
#ifndef SONG_H
#define SONG_H

#include <string>

class Song {
private:
    std::string title;
    std::string artist;
    int year;

public:
    Song(const std::string &t, const std::string &a, int y);
    void print() const;
    bool is_90s() const;
};

#endif
```

song.cpp — the definitions:

```
#include "song.h"
#include <iostream>

Song::Song(const std::string &t, const std::string &a, int y)
    : title(t), artist(a), year(y) {}
```

```cpp
void Song::print() const {
    std::cout << title << " by " << artist
              << " (" << year << ")"
              << std::endl;
}

bool Song::is_90s() const {
    return year >= 1990 && year <= 1999;
}
```

The `Song::` prefix tells the compiler that these functions belong to the `Song` class. This is the scope resolution operator `::` that you first saw with namespaces in Chapter 1.

The `#ifndef`/`#define`/`#endif` pattern is called an **include guard**. It prevents the header from being included more than once in the same compilation, which would cause duplicate definition errors.

Tip: Most compilers support `#pragma once` as a simpler alternative to include guards. It does the same thing and is commonly used, even though it is not part of the C++ standard.

`static` Members

Everything you have seen about classes so far has been about *instance* state: each `Song` has its own `title`, `artist`, and `year`, and each member function operates on one specific object via the hidden `this` pointer. But some things belong to the class as a whole, not to any single instance. How many `Song` objects have been created so far? What is the maximum title length the class allows? What is the default volume to play at when no volume is specified?

Here is what the "how many `Songs` have been created" question looks like *without* static members:

```cpp
// song_count.h
extern int song_count;

// song_count.cpp
int song_count = 0;

// song.h
#include <string>
#include "song_count.h"

class Song {
    std::string title;
public:
    Song(const std::string &t) : title(t) { ++song_count; }
    ~Song() { --song_count; }
};
```

The counter is conceptually part of the `Song` class — it only makes sense in relation to `Song` — but it lives outside the class as a global variable. That is awkward for several reasons:

- It needs an `extern` declaration in its own header and a definition in a `.cpp` file, just like any other shared global (you saw that pattern in Chapter 6).
- Anyone can modify `song_count` from anywhere in the program, defeating the encapsulation that classes are supposed to provide.
- The name `song_count` pollutes the global namespace.
- If you add a second piece of `Song`-wide state, you get yet another global with all the same problems.

`static` Data Members

The `static` keyword inside a class declaration gives you a variable that belongs to the *class* rather than to any instance:

```
// song.h
#include <string>

class Song {
    std::string title;
    static int count;           // declaration --- shared by all Songs
public:
    Song(const std::string &t) : title(t) { ++count; }
    ~Song() { --count; }

    static int instance_count() { return count; }
};
```

There is exactly one `count` for the whole program, no matter how many `Song` objects exist. Every `Song` constructor and destructor sees the same `count` and updates the same underlying storage.

There is one small wrinkle: declaring a `static` data member inside the class body is not enough — you also have to provide its *definition* somewhere, because the linker needs a place to put the actual storage. The convention is to declare it in the header and define it in the matching `.cpp` file:

```
// song.cpp
#include "song.h"

// definition --- exactly one, in one .cpp file
int Song::count = 0;
```

Notice the `Song::` prefix that attaches the definition to the class, and notice that the `static` keyword does **not** appear in the definition — it only appears in the declaration. Repeating `static` in the definition is a compile error.

If two `.cpp` files both define `Song::count`, the linker will complain about a duplicate definition for the same reason a plain global would. Static data members still obey the one-definition rule — the whole point is that they act like a well-behaved global that is scoped to the class instead of to the whole program.

Tip: For `static const` or `static constexpr` data members with a simple value, C++17 lets you define the value *inline* inside the class body and skip the separate `.cpp` definition entirely:

```
class Song {
    static constexpr int max_title_length = 200;
};
```

No matching `int Song::max_title_length = 200;` in a `.cpp` file is needed. This is the preferred style for class-wide constants.

`static` Member Functions

A `static` member function belongs to the class rather than to any instance. You call it by qualifying it with the class name and the scope resolution operator:

```
int n = Song::instance_count();
```

You do not need a `Song` object to call it — in fact, you do not need *any* `Songs` to exist at all.

A `static` member function has no `this` pointer, so it cannot touch non-static member variables or call non-static member functions directly. It can only see other `static` members. The `instance_count()` function

above works because `count` is also `static`.

Wut: The `static` keyword has two completely *unrelated* meanings depending on where you use it:

- **On a free (non-member) function** (Chapter 6): `static` gives the function **internal linkage**, meaning it is private to the `.cpp` file it lives in and invisible to the linker outside that file.
- **Inside a class declaration** (this chapter): `static` means the member belongs to the *class* rather than to an instance. It has nothing to do with linkage — a `static` member function is still visible to the linker like any other member function, and code in other translation units can call `Song::instance_count()` perfectly well.

Same keyword, completely different jobs. The C++ committee has been apologizing for this ever since.

When to Use `static` Members

Static members are the right tool when the data or behavior really is class-wide, not per-instance:

- **Class-wide constants and configuration** — maximum length, default values, version numbers. Prefer `static constexpr` when the value is known at compile time.
- **Class-wide counters or caches** — "how many of these have been created," "the most recently created one," a shared lookup table.
- **Factory functions** — static member functions that create and return new instances of the class, often as an alternative to constructors when the construction process is more involved than just initializing members.
- **Sentinel values** — a well-known constant that callers can compare against, like "not found."

Examples of good use from the standard library:

- `std::numeric_limits<int>::max()` — a `static constexpr` function returning the largest value an `int` can hold. A compile-time class-wide constant that depends only on the type, so attaching it to the type is exactly right.
- `std::string::npos` — a `static const` data member that `std::string::find()` returns to mean "not found." A sentinel value that is conceptually tied to `std::string` and nothing else.
- `std::chrono::system_clock::now()` — a `static` factory function that returns the current time as a `time_point`. The class *is* the clock; `now()` is a class-wide operation that manufactures an instance.

When Not to Use `static` Members

Static members get tempting as a way to bolt loose functions or global variables onto an existing class, and that temptation usually leads to bad design.

- **Do not create a class just to hold a bunch of static functions.** If none of the functions touch instance state, they are really free functions and should live in a namespace instead. A "utility class" that is never instantiated is almost always a namespace wearing a costume.
- **Do not use a static data member as a stealth global variable.** If the data is not conceptually tied to the class as a whole — if it just happens to be used by some of the class's methods — it is really a global, and all the usual problems with globals still apply (hidden coupling, hard-to-test code, surprising initialization order across translation units).
- **Do not mark a function `static` just because its current implementation happens not to touch `this`.** If the function logically operates on an instance, leave it as a normal member function. Otherwise, you pin callers to the `Class::foo()` syntax, and having to switch back to `obj.foo()` later when the implementation starts needing `this` is an annoying breaking change.

An example from the standard library that illustrates the temptation: `std::thread::hardware_concurrency()` reports how many hardware threads the system can run concurrently. It lives on `std::thread` as a static

method, but it has nothing to do with any particular thread — it is a report about the machine. You could equally well imagine it as a free function in `<thread>` alongside the other threading utilities. It is a reasonable design and not really a bug, but it also reads like the kind of function that ends up as a `static` member more by association than by necessity. When you are choosing between "static member function on `X`" and "free function in a namespace near `X`," lean toward the free function unless the operation is genuinely tied to the class.

Operator Overloading

C++ lets you define what operators like `==`, `+`, and others mean for your own types. This is called **operator overloading**. You write an operator overload as a member function named `operator` followed by the symbol.

The + Operator

You can overload `+` to make it do something meaningful for your class. Here is a `Playlist` where `+` adds a song and returns a new playlist:

```
class Playlist {
private:
    std::string name;
    std::vector<std::string> songs;

public:
    Playlist(const std::string &n) : name(n) {}

    Playlist operator+(const std::string &song) const {
        Playlist result = *this;
        result.songs.push_back(song);
        return result;
    }

    Playlist &operator+=(const std::string &song) {
        songs.push_back(song);
        return *this;
    }

    void print() const {
        std::cout << name << ":" << std::endl;
        for (size_t i = 0; i < songs.size(); ++i) {
            std::cout << "  " << i + 1 << ". " << songs[i] << std::endl;
        }
    }
};

Playlist p("90s Jams");
p = p + "Torn";
p += "Kiss from a Rose";
p += "Basket Case";
p.print();
```

Output:

```
90s Jams:
  1. Torn
  2. Kiss from a Rose
  3. Basket Case
```

The `operator+` member function takes a song title, makes a copy of the current playlist with `*this`, appends the song to the copy, and returns it as a new `Playlist`. The original playlist is not modified — just like `3 + 2` does not change the value of `3`.

The `operator+=` member function modifies the playlist in place and returns a reference to `*this`. Notice that `operator+` is `const` but `operator+=` is not. `operator+` does not modify the object it is called on — it creates a new playlist with the extra song — so it can be `const`. `operator+=` modifies the object itself by appending directly to its `songs` vector, so it cannot be `const`. This mirrors how built-in types work: `x + 1` does not change `x`, but `x += 1` does.

The == Operator for Comparison

You can also define what it means for two objects to be equal:

```
class Playlist {
    // ... (same members as above)

public:
    // ... (same constructor and methods as above)

    bool operator==(const Playlist &other) const {
        return name == other.name && songs == other.songs;
    }
};
```

```
Playlist a("Mix");
a = a + "Torn";
Playlist b("Mix");
b = b + "Torn";
Playlist c("Mix");
c = c + "Basket Case";

std::cout << (a == b) << std::endl;  // 1 (true)
std::cout << (a == c) << std::endl;  // 0 (false)
```

Notice that `operator==` has access to private members of `other` because `other` is the same class.

Wut: You can overload the function-call operator `()` as a member function. This makes an object *callable*, as if it were a function:

```
struct Greeter {
    std::string greeting;

    void operator()(const std::string &name) const {
        std::cout << greeting << ", " << name << "!" << std::endl;
    }
};

Greeter hola{"Hola"};
hola("amigo");       // prints: Hola, amigo!
```

An object that overloads `()` is called a **functor**. You will see in *Gorgo Continuing C++* that a lambda is really just a functor the compiler generates for you.

Conversion Operators

Just as a constructor can convert *to* your type, a **conversion operator** lets your type convert *to* another type. You write it as a member function named `operator` followed by the target type, with no explicit return type:

```
class Volume {
private:
    int level;

public:
    explicit Volume(int l) : level(l) {}

    operator int() const {
        return level;
    }
};
```

Now a `Volume` can be used anywhere an `int` is expected:

```
Volume v(11);
int n = v;                      // implicit conversion: n is 11
std::cout << v + 1 << std::endl; // 12
```

This is convenient, but implicit conversions can cause the same surprises as non-explicit constructors. The fix is the same — add `explicit`:

```
explicit operator int() const {
    return level;
}
```

Now the conversion only happens when you ask for it:

```
Volume v(11);
// int n = v;                   // ERROR: no implicit conversion
int n = static_cast<int>(v);    // OK: explicit cast
```

The most common use of `explicit` conversion operators is `explicit operator bool()`. This is the pattern the standard library uses for streams — it is why `if (std::cin)` works (a boolean context like `if` allows explicit conversions) but `int x = std::cin` does not:

```
class Connection {
private:
    int fd;

public:
    explicit Connection(int f) : fd(f) {}

    explicit operator bool() const {
        return fd >= 0;
    }
};

Connection conn(3);
// OK: bool context allows explicit conversion
if (conn) {
    std::cout << "connected" << std::endl;
}
// bool b = conn;                // ERROR: not a bool context
```

Tip: Prefer `explicit` on conversion operators. An implicit `operator bool()` would let code like `int x = conn + 5;` compile silently — `conn` would convert to `bool` (which is `true`, i.e., `1`), then `1 + 5` gives `6`. That is almost never what you want.

Putting It All Together

Here is a complete program that uses everything from this chapter:

```
#include <iostream>
#include <string>
#include <vector>

class Playlist {
private:
    std::string name;
    std::vector<std::string> songs;

public:
    Playlist(const std::string &n) : name(n) {}

    void add(const std::string &song) {
        songs.push_back(song);
    }

    Playlist operator+(const std::string &song) const {
        Playlist result = *this;
        result.songs.push_back(song);
        return result;
    }

    Playlist &operator+=(const std::string &song) {
        songs.push_back(song);
        return *this;
    }

    bool operator==(const Playlist &other) const {
        return name == other.name && songs == other.songs;
    }

    void print() const {
        std::cout << name << ":" << std::endl;
        for (size_t i = 0; i < songs.size(); ++i) {
            std::cout << "  " << i + 1 << ". " << songs[i] << std::endl;
        }
    }
};

int main()
{
    Playlist a("90s Jams");
    a.add("Torn");
    a += "Kiss from a Rose";

    Playlist b = a + "Basket Case";

    a.print();
    std::cout << std::endl;
    b.print();
    std::cout << std::endl;
```

```
    if (a == b) {
        std::cout << "Same playlist" << std::endl;
    } else {
        std::cout << "Different playlists" << std::endl;
    }

    return 0;
}
```

Output:

```
90s Jams:
  1. Torn
  2. Kiss from a Rose

90s Jams:
  1. Torn
  2. Kiss from a Rose
  3. Basket Case

Different playlists
```

Key Points

- A class bundles data and the functions that operate on that data together.
- Members are `private` by default in a `class` and `public` by default in a `struct`.
- Access specifiers (`public`, `private`, `protected`) control who can access members.
- Constructors initialize objects; use member initializer lists for efficiency.
- Mark single-argument constructors `explicit` to prevent implicit conversions.
- Destructors clean up when an object is destroyed.
- Mark member functions `const` when they do not modify the object.
- The `this` pointer refers to the current object inside a member function.
- Function overloading lets you define multiple functions with the same name but different parameter lists.
- Default parameters let callers omit trailing arguments by providing default values.
- Do not mix overloading and default parameters in ways that create ambiguous calls.
- Split large classes into a header file (`.h`) for declarations and a source file (`.cpp`) for definitions.
- Operator overloading lets you define the behavior of operators for your own types.
- Conversion operators (`operator T()`) let a class convert to another type; mark them `explicit` to prevent surprises.
- The `friend` keyword grants outside functions or classes access to private members — see Chapter 14.

Exercises

1. What is the difference between a `struct` and a `class` in C++? Why would you choose one over the other?
2. What does the following program print?

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

class Band {
private:
    std::string name;
```

```
    int formed;

public:
    Band(const std::string &n, int y) : name(n), formed(y) {
        std::cout << name << " arrives" << std::endl;
    }

    ~Band() {
        std::cout << name << " exits" << std::endl;
    }
};

int main()
{
    Band a("Metallica", 1981);
    Band b("Soundgarden", 1984);
    std::cout << "show time" << std::endl;
    return 0;
}
```

3. What is wrong with the following class?

```
class Counter {
private:
    int count;

public:
    Counter() : count(0) {}

    void increment() const {
        count++;
    }

    int get_count() const { return count; }
};
```

4. Why should you prefer member initializer lists over assignment in the constructor body? Give an example of a situation where the initializer list is *required*.

5. If a class has three `int` members and a `std::string` member, how many bytes *minimum* does an object of that class occupy on a system where `int` is 32 bits and `std::string` is 32 bytes? (Ignore padding for this question.)

6. What does the following code output?

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

class Song {
private:
    std::string title;
public:
    Song(const std::string &t) : title(t) {}

    bool operator==(const Song &other) const {
        return title == other.title;
    }
```

```
};

int main()
{
    Song a("All Star");
    Song b("All Star");
    Song c("Enter Sandman");

    std::cout << (a == b) << std::endl;
    std::cout << (a == c) << std::endl;
    return 0;
}
```

7. What is the bug in this code?

```
class Player {
private:
    std::string name;
    int score;

public:
    Player(const std::string &name, int score) {
        name = name;
        score = score;
    }
};
```

8. What does the following program print?

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

class Radio {
public:
    void play(const std::string &song) {
        std::cout << "Playing: " << song << std::endl;
    }

    void play(const std::string &song, int volume) {
        std::cout << "Playing: " << song << " at volume " << volume << std::endl;
    }

    void play(int station) {
        std::cout << "Tuned to station " << station << std::endl;
    }
};

int main()
{
    Radio r;
    r.play("Torn");
    r.play(98);
    r.play("Basket Case", 11);
    return 0;
}
```

9. What is wrong with the following code?

```
class Speaker {
public:
    void set_volume(int v) {
        volume = v;
    }

    void set_volume(int v, int max = 100) {
        volume = (v > max) ? max : v;
    }

private:
    int volume;
};
```

10. Why must default parameters appear at the end of the parameter list? What happens if you try to put a default parameter before a non-default one?

11. What is wrong with this code?

```
class TrackNumber {
public:
    int number;
    TrackNumber(int n) : number(n) {}
};

void play(TrackNumber t) {
    std::cout << "Track " << t.number << std::endl;
}

int main() {
    play(7);
    return 0;
}
```

12. What does `explicit operator bool()` allow that a non-explicit `operator bool()` also allows? What does it prevent?

13. Write a class called `Counter` with a private `int count` that starts at 0. Give it an `increment()` method, a `reset()` method, and a `const` method `value()` that returns the current count. Overload `operator==` to compare two counters by their count. Test it in `main()` by incrementing, printing, resetting, and comparing two counters.

14. **Where is the bug?**

```
class Playlist {
    std::vector<std::string> tracks;
public:
    int size() const { return tracks.size(); }
};

int main()
{
    Playlist p;
    p.tracks.push_back("Wonderwall");
    std::cout << p.size() << "\n";
```

```
    return 0;
}
```

What does the compiler say, and which design rule does it enforce? What is the smallest change you can make to `Playlist` so the program compiles, and what is the *better* change?

15. **Write a program** that defines a class `Builder` whose `add(int)` method returns `*this` by reference so calls can be chained:

    ```
    Builder b;
    b.add(1).add(2).add(3).add(4);
    ```

    The class should keep a `std::vector<int>` internally and have a `print()` method that prints all of the values added so far. Why does `add` return `*this` and not just a fresh `Builder`?

16. **Think about it:** A class has this conversion operator:

    ```
    class Volume {
        int level;
    public:
        explicit Volume(int v) : level(v) {}
        explicit operator bool() const { return level > 0; }
    };
    ```

    Why is `operator bool` marked `explicit`? Which of the following calls compile, and which do not?

    ```
    Volume v(3);
    if (v) { /* ... */ }                    // (a)
    bool b = v;                             // (b)
    bool b2 = static_cast<bool>(v);         // (c)
    int  n  = v;                            // (d)
    ```

13. Memory Management

Every variable you have created so far lives on the **stack** — a region of memory that is managed automatically. When a variable goes out of scope, its memory is reclaimed for you. This is simple and reliable, but it has limits.

Sometimes you need memory that outlives the current scope, or you need to allocate a size that is not known until the program is running. For that, C++ gives you the **heap** — a larger region of memory that you control explicitly.

In this chapter, you will learn how heap memory works, why manual management is error-prone, and how modern C++ smart pointers make it safe and easy.

Stack vs. Heap

The **stack** is fast and automatic. Every time you declare a local variable, it goes on the stack. When the function returns, all its stack variables are destroyed:

```
void play() {
    int volume = 11;  // lives on the stack
}  // volume is destroyed here
```

The **heap** (also called **free store**) is a separate pool of memory. You request memory from the heap at runtime, and it stays allocated until you explicitly release it — or until the program ends.

The stack is like a concert venue's coat check: you hand in your coat, get a ticket, and pick it up on the way out. The heap is like renting a storage unit: it is yours until you cancel the lease.

Why Not Just Use the Stack?

Stack memory has two limitations that come up in real programs.

The size must be known at compile time. If the user decides how many items to store, you cannot create a stack array for it:

```
void record_scores() {
    int count;
    std::cout << "How many scores? ";
    std::cin >> count;

    // int scores[count];  // NOT standard C++ --- size must be a constant
}
```

Trap: Most compilers will actually accept `int scores[count];` without complaint. This is a **variable-length array** (VLA), a feature from C99 that some C++ compilers support as an extension. Do not use it. VLAs were never part of the C++ standard, they allocate on the stack so a large `count` can overflow it and crash your program, and their support varies across compilers and flags. Use `std::vector<int>` (Chapter 8) or heap allocation instead.

The heap lets you allocate exactly as much memory as you need at runtime.

Stack variables die when the function returns. If a function creates an object and needs to hand it back to the caller, a stack variable will not work — it is destroyed before the caller can use it:

```
#include <string>

std::string* make_greeting() {
    std::string local = "Don't Speak";
```

```
    return &local;   // BUG: local is destroyed when the function returns
}                     // the caller gets a dangling pointer
```

The caller receives a pointer to memory that no longer exists. The heap solves this because heap memory persists until you explicitly free it.

Tip: Prefer stack allocation whenever possible. It is faster, automatic, and less error-prone. Only use the heap when you need memory to outlive the current scope or when the size is not known at compile time.

Pointers

Before you can work with the heap, you need to understand **pointers**. A pointer is a variable that holds the memory address of another variable. You have made it this far without needing pointers directly because references, smart pointers, and standard containers handle most situations. But `new` returns a pointer, so you need the basics.

Getting an Address

The **address-of operator** `&` gives you the memory address of a variable:

```
int volume = 11;
std::cout << &volume << std::endl;  // prints something like 0x7ffd3a2c
```

The exact number depends on where the operating system placed `volume` in memory.

Declaring a Pointer

A pointer variable is declared by putting `*` after the type. It stores an address, not a value:

```
int volume = 11;
int *ptr = &volume;   // ptr holds the address of volume
```

`int *ptr` reads as "ptr is a pointer to an int."

You can also have a pointer to a pointer:

```
int **pptr = &ptr;   // pptr holds the address of ptr
```

This comes up when you need to modify a pointer itself through a function, or when dealing with arrays of pointers (like `argv` from Chapter 1, which is really `char **`).

Dereferencing

To access the value a pointer points to, use the **dereference operator** `*`:

```
int volume = 11;
int *ptr = &volume;

std::cout << *ptr << std::endl;   // prints 11 --- the value at the address

*ptr = 5;                         // changes volume through the pointer
std::cout << volume << std::endl;  // prints 5
```

Wut: The `*` symbol does three different things depending on context. In a declaration like `int *ptr`, it means "pointer to." In an expression like `*ptr`, it means "dereference." In an expression like `a * b`, it means multiplication. The compiler always knows which is which from context, even if the reader has to think for a moment.

The Arrow Operator

When you have a pointer to a structure or class, you need to dereference it before accessing a member. The parentheses are required because `.` has higher precedence than `*`:

```
struct Song {
    std::string title;
    int year;
};

Song s = {"Popular", 1996};
Song *ptr = &s;

std::cout << (*ptr).title << std::endl;  // works but awkward
```

Because `(*ptr).member` is tedious to write, C++ provides the **arrow operator** `->` as a shorthand:

```
std::cout << ptr->title << std::endl;   // same thing, much cleaner
std::cout << ptr->year << std::endl;
```

`ptr->member` is exactly equivalent to `(*ptr).member`. You will see `->` everywhere in C++ — it is the standard way to access members through a pointer.

nullptr

A pointer that does not point to anything should be set to `nullptr`:

```
int *ptr = nullptr;   // points to nothing
```

Historically C++ used `NULL` to indicate a pointer to nothing. In C++, `NULL` is literally the integer `0`, which is recognized as an invalid address. C still uses `NULL`, and many older C++ code bases do too, but `nullptr` is preferred in modern C++ because it can be distinguished from an `int`. For example:

```
void look_up(int index);
void look_up(void *addr);
```

You would expect `look_up(NULL)` to invoke `void look_up(void *addr);` however, since `NULL` is the integer `0`, it matches the first `look_up` instead. `look_up(nullptr)` unambiguously calls the pointer overload.

Dereferencing a null pointer is undefined behavior — your program will almost certainly crash. Always check before dereferencing a pointer you are not sure about:

```
if (ptr != nullptr) {
    std::cout << *ptr << std::endl;
}
```

Tip: Modern C++ reduces the need for raw pointers significantly. References (Chapter 6) are safer for "point to an existing object" cases, and smart pointers (covered later in this chapter) are safer for heap memory. Raw pointers still appear in older code, C library interfaces, and `argv`, so you need to recognize them.

`new` and `delete`

The `new` operator allocates memory on the heap and returns a pointer to it. The `delete` operator releases that memory:

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

```cpp
int main()
{
    std::string *song = new std::string("Under the Bridge");
    std::cout << *song << std::endl;
    delete song;  // free the memory

    return 0;
}
```

Output:

```
Under the Bridge
```

After `delete`, the pointer `song` still exists, but the memory it points to has been freed. Using the pointer after `delete` is **undefined behavior**.

`new[]` and `delete[]` for Arrays

To allocate an array on the heap, use `new[]` and `delete[]`:

```cpp
int *scores = new int[5];  // allocate array of 5 ints

scores[0] = 10;
scores[1] = 20;
// ...

delete[] scores;  // free the array
```

Trap: If you allocate with `new[]`, you *must* free with `delete[]`. Using plain `delete` on an array allocated with `new[]` is undefined behavior. The compiler will not warn you — it will just silently corrupt memory.

Memory Leaks and Dangling Pointers

Manual memory management with `new` and `delete` is notoriously error-prone. Two of the most common bugs are **memory leaks** and **dangling pointers**.

Memory Leaks

A **memory leak** occurs when you allocate memory but never free it:

```cpp
void leak() {
    std::string *s = new std::string("Nothing Compares 2 U");
    // oops --- we never delete s
}  // s is destroyed, but the string on the heap lives on
```

Every time `leak()` is called, it allocates memory that is never released. Over time, your program eats more and more memory until it runs out.

Dangling Pointers

A **dangling pointer** is a pointer that refers to memory that has already been freed:

```cpp
int *p = new int(42);
delete p;
std::cout << *p << std::endl;  // DANGER: p is dangling
```

Dereferencing a dangling pointer is undefined behavior. Your program might crash, print garbage, or appear to work fine — until it does not.

Trap: After `delete`, set the pointer to `nullptr` if you plan to keep the pointer variable around. This does not prevent all dangling pointer bugs, but it makes it easier to check if a pointer is valid.

These problems are why modern C++ strongly discourages using raw `new` and `delete`. The solution is smart pointers.

Smart Pointers

Smart pointers are objects that manage heap memory for you. When a smart pointer goes out of scope, it automatically deletes the memory it owns. This pattern is called **RAII** — Resource Acquisition Is Initialization. The idea is that a resource (like heap memory) is tied to an object's lifetime: acquired in the constructor, released in the destructor.

Smart pointers live in the `<memory>` header.

`std::unique_ptr`

A `std::unique_ptr` represents **sole ownership** of a heap-allocated object. Only one `unique_ptr` can own a given piece of memory at a time. When the `unique_ptr` is destroyed, the memory is automatically freed.

```
#include <iostream>
#include <memory>
#include <string>

int main()
{
    auto song =
        std::make_unique<std::string>("Don't Speak");
    std::cout << *song << std::endl;

    // no delete needed --- memory is freed when song goes out of scope
    return 0;
}
```

Output:

```
Don't Speak
```

Its signature is:

```
std::unique_ptr<T> make_unique(Args... args);
```

`std::make_unique<T>(args...)` allocates a new `T` on the heap, passes `args` to its constructor, and wraps the result in a `unique_ptr`. Always prefer `make_unique` over `new`.

Because ownership is exclusive, you cannot copy a `unique_ptr`:

```
std::unique_ptr<int> a = std::make_unique<int>(42);
std::unique_ptr<int> b = a;  // ERROR: cannot copy a unique_ptr
```

But you can **move** it (more on this shortly):

```
std::unique_ptr<int> a = std::make_unique<int>(42);
std::unique_ptr<int> b = std::move(a);  // OK: ownership transferred to b
// a is now empty (nullptr)
```

Tip: `std::unique_ptr` should be your default choice for heap allocation. It has zero overhead compared to a raw pointer — the compiler generates the same code, but with automatic cleanup.

`std::shared_ptr`

Sometimes multiple parts of your code need to share ownership of the same object. A `std::shared_ptr` uses **reference counting** to track how many `shared_ptr`s point to the same memory. The memory is freed only when the last `shared_ptr` owning it is destroyed.

You create a `shared_ptr` with `std::make_shared`, whose signature mirrors `make_unique`:

```
std::shared_ptr<T> make_shared(Args... args);
```

Two useful member functions for inspecting and managing a `shared_ptr` are `use_count()` and `reset()`:

```
long use_count() const;   // returns the current reference count
void reset();             // releases this shared_ptr's ownership
```

```
#include <iostream>
#include <memory>
#include <string>

int main()
{
    auto song1 =
        std::make_shared<std::string>("Under the Bridge");
    // both point to the same string
    std::shared_ptr<std::string> song2 = song1;

    std::cout << *song1 << std::endl;
    std::cout << *song2 << std::endl;
    std::cout << "ref count: " << song1.use_count() << std::endl;

    song1.reset();  // song1 gives up ownership
    std::cout << "ref count: " << song2.use_count() << std::endl;

    // memory is freed when song2 goes out of scope
    return 0;
}
```

Output:

```
Under the Bridge
Under the Bridge
ref count: 2
ref count: 1
```

`std::make_shared` is the preferred way to create a `shared_ptr`, just as `make_unique` is for `unique_ptr`.

Tip: Use `shared_ptr` only when you truly need shared ownership. If one owner is enough, use `unique_ptr` instead — it is simpler and has no reference-counting overhead.

Getting a Raw Pointer from a Smart Pointer

Sometimes you need to pass a raw pointer to a function that does not understand smart pointers — a C library function, for example. Both `std::unique_ptr` and `std::shared_ptr` provide a `.get()` method that returns the raw pointer without releasing ownership:

```
T* get() const;
```

```
auto song = std::make_unique<std::string>("Under the Bridge");

// pass the raw pointer to a function that expects std::string*
std::string *raw = song.get();
std::cout << *raw << std::endl;   // "Under the Bridge"

// song still owns the memory --- do NOT delete raw
```

Trap: Never `delete` a pointer obtained from `.get()`. The smart pointer still owns the memory and will `delete` it when it goes out of scope. Deleting it yourself causes a double-free bug.

Move Semantics

When you copy a large object — like a `std::string` with a long value — the program has to duplicate all the data. **Move semantics** offer an alternative: instead of copying the data, you *transfer* it from one object to another, leaving the source in a valid but empty state.

Think of it like giving someone your notebook instead of photocopying every page.

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

int main()
{
    std::string a = "Nothing Compares 2 U";
    std::cout << "a: " << a << std::endl;

    std::string b = std::move(a);  // move a's contents into b
    std::cout << "b: " << b << std::endl;
    std::cout << "a: " << a << std::endl;  // a is now empty

    return 0;
}
```

Output:

```
a: Nothing Compares 2 U
b: Nothing Compares 2 U
a:
```

After the move, `a` is in a valid but unspecified state — for `std::string`, that means it is empty. The actual string data was not copied; ownership of the internal buffer was transferred to `b`.

Trap: After moving from an object, do not use it unless you assign a new value to it first. The object is in a valid state, but its contents are unspecified.

Its simplified signature is:

```
T&& move(T&& value);
```

`std::move` does not actually move anything — it simply casts its argument to an **rvalue reference**, which tells the compiler "it is OK to move from this." The actual moving is done by the receiving object's move constructor or move assignment operator. You will learn about the special member functions (copy constructor, move constructor, and their assignment counterparts) in Chapter 14.

Putting It All Together

Here is a complete program that demonstrates smart pointers and move semantics:

```
#include <iostream>
#include <memory>
#include <string>

class Song {
private:
    std::string title;
    std::string artist;

public:
    Song(const std::string &t, const std::string &a) : title(t), artist(a) {
        std::cout << "  created: " << title << std::endl;
    }

    ~Song() {
        std::cout << "  destroyed: " << title << std::endl;
    }

    void print() const {
        std::cout << "  " << title << " by " << artist << std::endl;
    }
};

int main()
{
    std::cout << "--- unique_ptr ---" << std::endl;
    {
        auto song = std::make_unique<Song>("Don't Speak", "No Doubt");
        song->print();
    }  // song is destroyed here
    std::cout << std::endl;

    std::cout << "--- shared_ptr ---" << std::endl;
    {
        std::shared_ptr<Song> s1;
        {
            auto s2 = std::make_shared<Song>("Under the Bridge", "RHCP");
            s1 = s2;
            std::cout << "  ref count: " << s1.use_count() << std::endl;
        }  // s2 destroyed, but Song lives on
        std::cout << "  ref count: " << s1.use_count() << std::endl;
        s1->print();
    }  // s1 destroyed, Song is finally freed
    std::cout << std::endl;
```

```
    std::cout << "--- move ---" << std::endl;
    std::string lyrics = "Nada se compara contigo";
    std::cout << "  before: " << lyrics << std::endl;
    std::string moved = std::move(lyrics);
    std::cout << "  moved:  " << moved << std::endl;
    std::cout << "  after:  " << lyrics << std::endl;

    return 0;
}
```

Output:

```
--- unique_ptr ---
  created: Don't Speak
  Don't Speak by No Doubt
  destroyed: Don't Speak

--- shared_ptr ---
  created: Under the Bridge
  ref count: 2
  ref count: 1
  Under the Bridge by RHCP
  destroyed: Under the Bridge

--- move ---
  before: Nada se compara contigo
  moved:  Nada se compara contigo
  after:
```

Key Points

- A **pointer** holds the address of another variable. Use `&` to get an address, `*` to dereference, and `->` to access members through a pointer.
- The **stack** is fast and automatic; the **heap** requires manual management.
- `new` allocates on the heap; `delete` frees it. Use `new[]`/`delete[]` for arrays.
- Forgetting `delete` causes **memory leaks**. Using a pointer after `delete` creates a **dangling pointer**.
- **`std::unique_ptr`** provides sole ownership with automatic cleanup — use it as your default.
- **`std::shared_ptr`** provides shared ownership via reference counting — use it when multiple owners are needed.
- Always prefer `std::make_unique` and `std::make_shared` over raw `new`.
- **Move semantics** transfer resources instead of copying them, which is more efficient.
- **RAII** ties resource lifetimes to object lifetimes — acquire in the constructor, release in the destructor.

Exercises

1. What is the difference between stack and heap memory? Give one situation where you would need to use the heap.
2. What does the following program print?

```
#include <iostream>
#include <memory>

int main()
{
```

```
    auto p = std::make_shared<int>(99);
    auto q = p;
    auto r = p;

    std::cout << p.use_count() << std::endl;

    q.reset();
    std::cout << p.use_count() << std::endl;

    r.reset();
    std::cout << p.use_count() << std::endl;

    return 0;
}
```

3. What is the bug in the following code?

```
void play() {
    int *volumes = new int[3];
    volumes[0] = 7;
    volumes[1] = 9;
    volumes[2] = 11;
    delete volumes;
}
```

4. Why can you not copy a `std::unique_ptr`? What should you do instead if you want to transfer ownership?

5. After `std::move(a)` is called, is it safe to use `a`? What state is `a` in?

6. What is wrong with the following code?

```
#include <memory>
#include <iostream>

int main()
{
    int *raw = new int(42);
    std::unique_ptr<int> a(raw);
    std::unique_ptr<int> b(raw);

    std::cout << *a << std::endl;
    std::cout << *b << std::endl;
    return 0;
}
```

7. If a `std::shared_ptr` is copied 4 times (so there are 5 `shared_ptr`s total pointing to the same object), what is the reference count? How many of those `shared_ptr`s need to be destroyed before the object is freed?

8. Write a program that creates a `std::unique_ptr<std::string>` holding your favorite 90s song title. Move it to a second `unique_ptr`, then print from the second and verify the first is empty (check with `if (!ptr)`).

9. What does the following code print?

```
int x = 10;
int *p = &x;
```

```
*p = 20;
std::cout << x << std::endl;
```

10. Given a `struct Song { std::string title; int year; };` and a pointer `Song *ptr`, write two equivalent expressions to access `title` — one using `*` and `.`, the other using `->`.

11. **Where is the bug?**

```
void play(Song *song)
{
    std::cout << song->title << " (" << song->year << ")\n";
}

int main()
{
    Song *s = nullptr;
    play(s);
    return 0;
}
```

    Why is this dangerous, and what is the smallest change to `play` that makes it safe?

12. **Think about it:** Explain RAII in your own words. Why is `std::unique_ptr` an RAII wrapper around `new`/`delete`? Name two other RAII types you have already seen earlier in this book.

13. **Where is the bug?**

```
void make_playlist()
{
    std::string *fav = new std::string("Wonderwall");
    if (fav->size() > 100) {
        return;
    }
    std::cout << *fav << "\n";
    delete fav;
}
```

    The function looks fine in the common case but leaks memory in one specific path. Identify the leak and rewrite the function so it cannot leak no matter which return path is taken (without sprinkling extra `delete` calls everywhere).

14. **Write a program** that uses `std::unique_ptr<int>` to wrap a heap integer and then passes the underlying raw pointer to a small C-style function

```
void c_api(int *p)
{
    *p += 1;
}
```

    Use `.get()` to obtain the raw pointer, call `c_api`, and then print the value. Why is it important that the `unique_ptr` *keeps* ownership across the call to `c_api` — in particular, why must `c_api` not call `delete` on its parameter?

14. Special Members and Friends

In Chapter 12 you learned how to define classes with constructors, destructors, and member functions. Those tools let you create well-behaved types, but two questions remain. First, when your class manages a resource like raw memory, how do you make sure copying, moving, and cleanup all work correctly? Second, how do you give an outside function or class access to private data when making it a member is not an option? This chapter covers both: the special member functions the compiler can generate (and the rules for writing your own), and the `friend` keyword for granting controlled access.

Special Member Functions and the Rule of Five

When you write a class, the compiler can automatically generate up to five special member functions:

1. **Destructor** — cleans up when the object is destroyed
2. **Copy constructor** — creates a new object as a copy of another
3. **Copy assignment operator** — replaces an existing object's contents with a copy of another
4. **Move constructor** — creates a new object by moving from another
5. **Move assignment operator** — replaces an existing object's contents by moving from another

If your class manages a resource (like raw heap memory), and you write any one of these five, you almost certainly need to write *all* five. This is called the **Rule of Five**.

Here is a simplified example of a class that manages its own heap memory. The example uses two C string functions from `<cstring>` whose signatures are:

```
size_t strlen(const char* str);    // returns length of a C string
char*  strcpy(char* dest, const char* src);  // copies src into dest
```

```
#include <iostream>
#include <cstring>

class Lyric {
private:
    char *text;

public:
    // constructor
    Lyric(const char *t) {
        text = new char[std::strlen(t) + 1];
        std::strcpy(text, t);
    }

    // destructor
    ~Lyric() {
        delete[] text;
    }

    // copy constructor
    Lyric(const Lyric &other) {
        text = new char[std::strlen(other.text) + 1];
        std::strcpy(text, other.text);
    }

    // copy assignment
    Lyric &operator=(const Lyric &other) {
        if (this != &other) {
```

```
            delete[] text;
            text = new char[std::strlen(other.text) + 1];
            std::strcpy(text, other.text);
        }
        return *this;
    }

    // move constructor
    Lyric(Lyric &&other) noexcept : text(other.text) {
        other.text = nullptr;
    }

    // move assignment
    Lyric &operator=(Lyric &&other) noexcept {
        if (this != &other) {
            delete[] text;
            text = other.text;
            other.text = nullptr;
        }
        return *this;
    }

    void print() const {
        if (text) {
            std::cout << text << std::endl;
        }
    }
};
```

Notice the `noexcept` keyword on the move constructor and move assignment operator. As you learned in Chapter 11, `noexcept` promises the compiler that these functions will not throw exceptions. Standard library containers like `std::vector` will only use your move operations (instead of slower copies) during reallocation if they are marked `noexcept`.

That is a lot of code just to manage a string. This brings us to better tools for controlling what the compiler generates.

Defaulted and Deleted Functions

The compiler's auto-generation rules create two practical problems. The first is that writing *any* constructor suppresses the default constructor, and writing a destructor or copy operation can suppress the move operations. Sometimes the compiler-generated versions are exactly what you want, but you have to write them by hand just to get them back. The second problem is the opposite: sometimes the compiler happily generates a function that makes no sense for your type. Imagine a class that represents an open connection to a hardware device — copying it would mean two objects fighting over the same device, but the compiler will generate a copy constructor anyway unless you stop it.

Before C++11, the workaround for the second problem was to declare the unwanted function `private` and never define it. That "worked," but anyone who accidentally called it got a confusing linker error instead of a clear explanation.

C++ gives you two tools to solve these problems: `= default` and `= delete`.

= default

When you write `= default`, you are saying "generate the default version of this function for me." This solves the first problem: adding one special member function suppresses the compiler's auto-generation of others, but you still want the default behavior:

```
class Song {
private:
    std::string title;
    std::string artist;

public:
    Song(const std::string &t, const std::string &a) : title(t), artist(a) {}

    // Writing a custom constructor suppresses the default constructor.
    // Bring it back:
    Song() = default;

    // The compiler-generated versions are fine for these:
    Song(const Song &) = default;
    Song &operator=(const Song &) = default;
    ~Song() = default;
};
```

`= default` also documents your intent: it tells anyone reading the code "I thought about this and the compiler's version is correct."

= delete

`= delete` solves the second problem. When you write `= delete`, the function exists but calling it is a compile-time error. This lets you prevent operations that do not make sense for your type:

```
class AudioStream {
private:
    int device_id;

public:
    AudioStream(int id) : device_id(id) {}

    // Copying an active audio stream would cause two objects
    // to fight over the same hardware device.
    AudioStream(const AudioStream &) = delete;
    AudioStream &operator=(const AudioStream &) = delete;

    // Moving is fine --- ownership transfers cleanly.
    AudioStream(AudioStream &&other) noexcept : device_id(other.device_id) {
        other.device_id = -1;
    }

    AudioStream &operator=(AudioStream &&other) noexcept {
        if (this != &other) {
            device_id = other.device_id;
            other.device_id = -1;
        }
        return *this;
    }
```

```
};

AudioStream a(1);
AudioStream b = a;              // ERROR: copy constructor is deleted
AudioStream c = std::move(a);   // OK: move constructor is available
```

You can delete any function, not just special members. A common use is preventing implicit conversions:

```
void set_volume(int v);
void set_volume(double) = delete;  // prevent set_volume(3.14)
```

Tip: `= delete` gives a clear compiler error with a message like "use of deleted function." This is much better than making a function private and leaving it undefined, which was the pre-C++11 workaround and produced cryptic linker errors instead.

The Rule of Zero

The **Rule of Zero** says: if your class does not manage a resource directly, do not write any of the five special member functions. Let the compiler generate them for you.

This is closely related to **RAII** (Resource Acquisition Is Initialization), a fundamental C++ pattern where you acquire resources in the constructor and release them in the destructor. When you follow the Rule of Zero, you rely on types that already implement RAII (like `std::string`, `std::vector`, and `std::unique_ptr`) so your class does not need to.

How do you avoid managing resources directly? Use smart pointers and standard library types that already handle their own memory:

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

class Lyric {
private:
    std::string text;  // std::string manages its own memory

public:
    Lyric(const std::string &t) : text(t) {}

    void print() const {
        std::cout << text << std::endl;
    }
};
```

This version does the same thing as the Rule of Five version above, but in a fraction of the code. The compiler-generated copy constructor, move constructor, and destructor all do the right thing because `std::string` already knows how to copy, move, and clean up after itself.

Tip: Follow the Rule of Zero whenever you can. Use `std::string` instead of `char*`, `std::vector` instead of raw arrays, and `std::unique_ptr` instead of raw `new`/`delete`. If all your members manage themselves, you do not need to write any special member functions.

Friends

So far, only a class's own member functions can access its private data. But sometimes an outside function or another class genuinely needs that access, and making it a member function is inconvenient or impossible.

Consider printing a `Playlist` with `std::cout`. You would like to write `std::cout << my_playlist`, but `operator<<` cannot be a member function of `Playlist` — the left operand of `<<` is a `std::ostream`, not a `Playlist`, so the compiler would need to add the overload to `std::ostream`, which you do not own. It has to be a free function, and a free function cannot access private members.

Or consider a class that manages another class's internals — like a `DJ` that manipulates a `Playlist`'s track order. You could make `DJ` a subclass or merge the two classes together, but neither makes sense: a DJ is not a kind of playlist, and a playlist does not need DJ behavior baked in.

C++ solves both problems with the `friend` keyword. A class can declare specific functions or classes as **friends**, granting them access to its private and protected members.

Friend Functions

To make a free function a friend, declare it with the `friend` keyword inside the class:

```
#include <iostream>
#include <string>
#include <vector>

class Playlist {
private:
    std::string name;
    std::vector<std::string> songs;

public:
    Playlist(const std::string &n) : name(n) {}

    void add(const std::string &song) {
        songs.push_back(song);
    }

    friend std::ostream &operator<<(std::ostream &os, const Playlist &p);
};

std::ostream &operator<<(std::ostream &os, const Playlist &p) {
    os << p.name << ":" << std::endl;
    for (size_t i = 0; i < p.songs.size(); ++i) {
        os << "  " << i + 1 << ". " << p.songs[i] << std::endl;
    }
    return os;
}

Playlist p("90s Jams");
p.add("I'll Be There for You");
p.add("Torn");
std::cout << p;
```

Output:

```
90s Jams:
  1. I'll Be There for You
  2. Torn
```

The `friend` declaration inside `Playlist` tells the compiler that `operator<<` may access `name` and `songs` directly, even though it is not a member function. The function itself is defined outside the class, just like any free function.

Notice that `operator<<` returns `std::ostream &` so that calls can be chained: `std::cout << a << b`. This is the same pattern the standard library uses for `std::cout << "hello" << std::endl` — each `<<` returns the stream, ready for the next one.

Friend Classes

You can also make an entire class a friend. Every member function of the friend class then has access to the private members:

```
class Playlist {
private:
    std::string name;
    std::vector<std::string> songs;

public:
    Playlist(const std::string &n) : name(n) {}

    void add(const std::string &song) {
        songs.push_back(song);
    }

    void print() const {
        std::cout << name << ":" << std::endl;
        for (size_t i = 0; i < songs.size(); ++i) {
            std::cout << "  " << i + 1 << ". " << songs[i] << std::endl;
        }
    }

    friend class DJ;
};

class DJ {
private:
    std::string name;

public:
    DJ(const std::string &n) : name(n) {}

    void intro(const Playlist &p) const {
        std::cout << name << ": up next, " << p.songs.size()
                  << " tracks from " << p.name << "!" << std::endl;
    }

    void swap_first_last(Playlist &p) const {
        if (p.songs.size() > 1) {
            std::string temp = p.songs.front();
            p.songs.front() = p.songs.back();
            p.songs.back() = temp;
        }
    }
};

Playlist p("90s Mix");
p.add("Torn");
p.add("Kiss from a Rose");
```

```
p.add("I'll Be There for You");

DJ dj("DJ Jazzy Jeff");
dj.intro(p);
p.print();
```

Output:

```
DJ Jazzy Jeff: up next, 3 tracks from 90s Mix!
90s Mix:
  1. Torn
  2. Kiss from a Rose
  3. I'll Be There for You
```

The `DJ` class can read and modify `Playlist`'s private `songs` and `name` because `Playlist` declared `DJ` as a friend. Notice that the friendship is one-directional — `DJ` can access `Playlist`'s private members, but `Playlist` cannot access `DJ`'s private members.

Rules of Friendship

Friendship has a few important rules:

- **Friendship is granted, not taken.** A class must declare its own friends inside its definition. You cannot claim friendship from outside.
- **Friendship is not mutual.** If `Playlist` declares `DJ` as a friend, `DJ` can access `Playlist`'s private members, but `Playlist` cannot access `DJ`'s private members unless `DJ` also declares `Playlist` as a friend.
- **Friendship is not inherited.** If `DJ` is a friend of `Playlist`, a class derived from `DJ` does not automatically get that friendship.
- **Friendship is not transitive.** If `A` is a friend of `B`, and `B` is a friend of `C`, `A` is *not* automatically a friend of `C`.

Tip: Use `friend` sparingly. Every friend is a piece of outside code that depends on your class's internal representation. If you change how the class stores its data, you have to update every friend too. Prefer member functions or public interfaces when possible, and reserve `friend` for cases like `operator<<` where there is no alternative.

Trap: Declaring too many friends defeats the purpose of making members private in the first place. If you find yourself adding friends frequently, consider whether the class's public interface is missing something.

Key Points

- The compiler can generate five special member functions: destructor, copy constructor, copy assignment, move constructor, and move assignment.
- The **Rule of Five**: if you write any one of the five, write all five.
- `= default` explicitly requests the compiler-generated version; `= delete` prevents a function from being called.
- Use `= delete` to make a type non-copyable, non-movable, or to prevent implicit conversions.
- The **Rule of Zero**: prefer types that manage themselves so you do not need to write any special member functions.
- **RAII** ties resource lifetimes to object lifetimes — acquire in the constructor, release in the destructor.
- The `friend` keyword grants a specific function or class access to private members.
- Use `friend` for operators like `<<` where the left operand is not your class.
- Friendship is granted, not taken; it is not mutual, inherited, or transitive.

Exercises

1. Explain the difference between the Rule of Five and the Rule of Zero. Which one should you prefer and why?

2. A coworker writes a class with a move constructor but `std::vector` keeps copying objects instead of moving them during reallocation. What is wrong with the move constructor?

```
class Track {
private:
    std::string title;
    std::vector<int> samples;

public:
    Track(const std::string &t) : title(t) {}

    Track(Track &&other)
        : title(std::move(other.title)),
          samples(std::move(other.samples)) {}
};
```

3. What does `= default` do when applied to a special member function? Why would you write `Song() = default;` instead of just omitting the default constructor?

4. What does the following code do, and why is it useful?

```
class Connection {
public:
    Connection(int fd) : fd_(fd) {}
    Connection(const Connection &) = delete;
    Connection &operator=(const Connection &) = delete;
private:
    int fd_;
};
```

5. Why does `operator<<` for output have to be a free function (or a friend) rather than a member function of your class?

6. What does the following program print?

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

class Vault {
private:
    std::string secret;

public:
    Vault(const std::string &s) : secret(s) {}

    friend void peek(const Vault &v);
};

void peek(const Vault &v) {
    std::cout << v.secret << std::endl;
}

int main()
```

```
{
    Vault v("Vogue");
    peek(v);
    return 0;
}
```

7. If class `A` declares class `B` as a friend, and class `B` declares class `C` as a friend, can `C` access `A`'s private members? Why or why not?

8. A class has a `std::string name`, a `std::vector<int> scores` with 3 elements, and an `int id`. How many of the five special member functions do you need to write if all members are standard library types or built-in types?

9. Write a class called `Album` with private members for `title` (string), `artist` (string), and `track_count` (int). Give it a parameterized constructor, a `const` member function that prints the album info, an overloaded `==` operator that compares all three fields, and a friend `operator<<` for output. Test it in `main()` by creating two albums, printing them with `<<`, and comparing them.

10. **Write a program** that defines a class `Buffer` that owns a heap-allocated `char *` and a `size_t` length. Implement **all five** special member functions explicitly:

    - destructor
    - copy constructor
    - copy assignment operator
    - move constructor (mark `noexcept`)
    - move assignment operator (mark `noexcept`)

    Add a small helper that prints which special member is running ("copy ctor", "move ctor", and so on) so you can watch them fire. In `main`, create one `Buffer`, copy it into another, move-construct a third, and let all of them go out of scope. Trace the output by hand before you run it and confirm it matches.

11. **Where is the bug?**

```
class Buffer {
    char *data;
    std::size_t len;
public:
    Buffer(std::size_t n) : data(new char[n]), len(n) {}
    ~Buffer() { delete[] data; }

    Buffer &operator=(const Buffer &other) {
        delete[] data;
        len = other.len;
        data = new char[len];
        for (std::size_t i = 0; i < len; ++i) data[i] = other.data[i];
        return *this;
    }
};

int main()
{
    Buffer b(100);
    b = b;            // assign to itself
    return 0;
}
```

    Walk through what happens inside `operator=` when the right-hand side *is* the left-hand side. Why is the result undefined behavior, and what is the standard fix?

12. **Think about it:** Why does `std::vector` insist that the move constructor and move assignment operator be marked `noexcept` before it will use them? What does the vector do *instead* if your move operations are not `noexcept`, and what is the performance cost?

15. Odds and Ends

You have come a long way. You can write programs with variables, control flow, functions, classes, containers, and file I/O. But there are practical gaps that come up in real programs. What if you need to terminate a program from deep inside a nested function call? What if you need to call a C library from C++? What if you need to convert between types safely, measure how long something takes, or generate random numbers? This chapter covers those remaining topics: `exit()` for program termination, `extern "C"` for C interoperability, C++ casting operators, the `<chrono>` time library, and the `<random>` library.

exit()

You already know that `return 0;` in `main()` ends the program successfully. But what if you need to stop the program from deep inside a function, not just from `main()`?

The `exit()` function, declared in `<cstdlib>`, terminates the program immediately from anywhere. Its signature is:

```
void exit(int status);
```

```
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>

void load_config(const std::string& filename)
{
    std::ifstream infile(filename);

    if (!infile) {
        std::cerr << "Fatal: could not open " << filename << std::endl;
        exit(EXIT_FAILURE);
    }

    std::cout << "Config loaded." << std::endl;
}

int main()
{
    load_config("settings.cfg");
    std::cout << "Program running..." << std::endl;

    return EXIT_SUCCESS;
}
```

If `settings.cfg` does not exist, the program prints an error and stops immediately. The line "Program running..." never executes.

`EXIT_SUCCESS` and `EXIT_FAILURE` are constants defined in `<cstdlib>`. `EXIT_SUCCESS` is typically `0` and `EXIT_FAILURE` is typically `1`, but using the named constants makes your intent clearer.

exit() vs return

When you call `return` from `main()`, the program exits in a controlled way — local variables in `main()` are cleaned up, destructors are called. `exit()` also performs cleanup: it flushes output streams and calls functions registered with `atexit()`. The signature of `atexit()` is:

```
int atexit(void (*func)());
```

However, local variables on the stack do not have their destructors called when `exit()` is used.

Tip: Prefer `return` from `main()` when possible. Use `exit()` when you need to terminate from a function deep in the call stack and returning an error code all the way up to `main()` would be impractical.

Trap: Because `exit()` does not call destructors for local variables on the stack, resources managed by RAII (like file handles or smart pointers in local scope) may not be cleaned up properly. Use `exit()` sparingly and with awareness of this limitation.

extern "C"

C++ grew out of C, and there is a massive amount of existing C code in the world. Sometimes you need to call C functions from C++, or make C++ functions callable from C. The key to this is `extern "C"`.

Name Mangling

When you write a function in C++, the compiler does not store the function name as-is in the compiled output. Instead, it **mangles** the name — it encodes the function name along with its parameter types into a unique symbol.

This is necessary because C++ supports function overloading. Consider:

```
void play(int track);
void play(const std::string& song);
void play(int track, bool repeat);
```

All three functions are named `play`, but they take different parameters. The compiler needs to tell them apart in the compiled output, so it might mangle them into something like `_Z4playi`, `_Z4playRKSs`, and `_Z4playib`. The exact mangled names depend on the compiler, but the point is that each overload gets a unique name.

C does not have function overloading, so C compilers do not mangle names. A C function called `play` is simply stored as `play` in the compiled output.

This creates a problem: if you try to call a C function from C++, the C++ compiler will look for a mangled name that does not exist.

Disabling Name Mangling

`extern "C"` tells the C++ compiler: "do not mangle this function name — use C-style naming."

```
extern "C" void c_function();
```

Now the C++ compiler knows to look for `c_function` without mangling, matching what a C compiler would produce.

Calling C Functions from C++

Imagine you have a C library with a function you want to use. You would declare it with `extern "C"`:

```
#include <iostream>

// Tell C++ this function uses C naming conventions
extern "C" {
    double sqrt(double x);
    int abs(int n);
}
```

```cpp
int main()
{
    std::cout << "La copa de la vida: sqrt(1998) = "
              << sqrt(1998) << std::endl;
    std::cout << "abs(-1) = " << abs(-1) << std::endl;

    return 0;
}
```

The `extern "C"` block tells the compiler that all functions declared inside it use C linkage.

Tip: In practice, you rarely need to write `extern "C"` declarations yourself. Most C library headers already handle this. But understanding why it exists helps you debug linker errors when mixing C and C++ code.

Wrapping C Headers

If you write a header that needs to work in both C and C++ code, you can use a common pattern:

```cpp
#ifdef __cplusplus
extern "C" {
#endif

void mi_funcion(int x);
int otra_funcion(const char* s);

#ifdef __cplusplus
}
#endif
```

`__cplusplus` is a macro that is defined only when compiling with a C++ compiler. When compiled as C++, the functions get `extern "C"` linkage. When compiled as C, the `extern "C"` parts are skipped entirely because C does not understand that syntax.

Wut: `extern "C"` does not mean "compile this as C code." The code inside `extern "C"` is still C++ — you can use C++ features. It only affects how the function name is stored in the compiled output.

Numbers and Casting

Everything is a Number

To the CPU, there are no strings, no classes, no booleans. There are only numbers — sequences of bits stored in memory. The types you use in C++ tell the compiler how to interpret those bits.

An `int` is a number you want to do arithmetic with. A `char` is also a number — just a smaller one. When you write `'A'`, the compiler stores the number `65`. When you write `'0'`, it stores `48`.

```cpp
#include <iostream>

int main()
{
    char letter = 'A';
```

```
    std::cout << "As char: " << letter << std::endl;
    std::cout << "As int:  " << static_cast<int>(letter) << std::endl;
    std::cout << "'A' + 1 = " << static_cast<char>(letter + 1) << std::endl;

    return 0;
}
```

Output:

```
As char: A
As int:  65
'A' + 1 = B
```

The same bits that represent the character `'A'` also represent the integer `65`. The type is just a label that tells the compiler what to do with the number.

Bit Widths and Ranges

Different types use different numbers of bits, which determines the range of values they can hold.

Type	Bits	Minimum	Maximum
`int8_t`	8	-128	127
`uint8_t`	8	0	255
`int16_t`	16	-32,768	32,767
`uint16_t`	16	0	65,535
`int32_t`	32	-2,147,483,648	2,147,483,647
`uint32_t`	32	0	4,294,967,295

These fixed-width types from `<cstdint>` guarantee exactly how many bits they use. The regular `int` is at least 16 bits but usually 32 bits on modern systems.

Trap: If you store a value too large for a type, it wraps around. For `uint8_t`, `255 + 1` becomes `0`. For `int8_t`, `127 + 1` becomes `-128`. This is a common source of subtle bugs.

C++ Casts

Sometimes you need to convert a value from one type to another. This is called **casting**. C++ provides four named cast operators that are safer and more expressive than the old C-style cast.

static_cast `static_cast` is the most common cast. Its syntax is:

```
static_cast<new_type>(expression)
```

Use it for well-defined, compile-time conversions between related types.

```
double pi = 3.14159;
int truncated = static_cast<int>(pi);  // 3 --- decimal part is lost

int score = 98;
double pct = static_cast<double>(score) / 100;  // 0.98
```

This is the cast you will use most often. It handles numeric conversions, conversions between related pointer types in a class hierarchy, and other conversions the compiler can verify at compile time.

dynamic_cast `dynamic_cast` is used for safe downcasting in class hierarchies with virtual functions. Its syntax is:

```
dynamic_cast<new_type>(expression)
```

It checks at runtime whether the cast is valid.

```
#include <iostream>

class Base {
public:
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    void special() { std::cout << "Do you believe?" << std::endl; }
};

int main()
{
    Base* bp = new Derived();
    Derived* dp = dynamic_cast<Derived*>(bp);

    if (dp != nullptr) {
        dp->special();
    } else {
        std::cout << "Cast failed" << std::endl;
    }

    delete bp;
    return 0;
}
```

If the object pointed to by `bp` is not actually a `Derived`, `dynamic_cast` returns `nullptr` instead of producing undefined behavior.

Tip: `dynamic_cast` only works with polymorphic types — classes that have at least one `virtual` function. If you have not studied inheritance yet, just know that this cast exists for safely converting between related class types at runtime.

const_cast `const_cast` adds or removes `const` from a pointer or reference. Its syntax is:

```
const_cast<new_type>(expression)
```

This is rarely needed and usually a sign that something in the design should be reconsidered.

```
#include <iostream>

void legacy_print(char* s)
{
    std::cout << s << std::endl;
}

int main()
{
```

```
    const char* song = "Believe";
    // legacy_print(song);  // Error: cannot convert const char* to char*
    legacy_print(const_cast<char*>(song));  // Compiles, but be careful

    return 0;
}
```

The main legitimate use is interfacing with old C APIs that take non-const pointers but promise not to modify the data.

Trap: If you use `const_cast` to remove `const` and then actually modify the data, the behavior is undefined if the original object was declared as `const`. Only use `const_cast` when you are certain the data will not be modified.

reinterpret_cast `reinterpret_cast` tells the compiler to treat the bits of one type as if they were another type entirely. Its syntax is:

```
reinterpret_cast<new_type>(expression)
```

This is the most dangerous cast and should be used rarely.

```
#include <iostream>
#include <cstdint>

int main()
{
    int value = 42;
    uintptr_t addr = reinterpret_cast<uintptr_t>(&value);

    std::cout << "Address of value: " << addr << std::endl;

    return 0;
}
```

This cast performs no conversion — it just reinterprets the bit pattern. It is used in low-level code like memory allocators or hardware interfaces.

Wut: `reinterpret_cast` does not change the bits at all. A `static_cast` from `float` to `int` actually converts the value (3.14 becomes 3). A `reinterpret_cast` would take the raw bits of the `float` and pretend they are an `int`, producing a completely different and probably meaningless number.

Why C++ Casts Over C-Style Casts?

In C (and in C++), you can cast with the syntax `(type)value`:

```
double pi = 3.14;
int n = (int)pi;  // C-style cast --- works but not recommended in C++
```

C++ also allows a **functional-style cast** that looks like a function call:

```
int n = int(pi);  // functional-style cast --- same thing, different syntax
```

Both forms are equivalent — `(int)pi` and `int(pi)` do exactly the same thing. Neither is recommended in new C++ code.

The problem with both forms is that they are blunt instruments. They can silently perform any of the four C++ casts, and you cannot tell which one just by looking at the code.

The C++ named casts are preferred because:

- **They express intent.** `static_cast` says "this is a safe, well-defined conversion." `reinterpret_cast` says "I know this is dangerous."
- **They are searchable.** You can search your codebase for `reinterpret_cast` to find all the dangerous casts. Good luck finding all the C-style casts with a search.
- **They are restrictive.** Each C++ cast only allows certain conversions. A C-style cast can do anything, including things you did not intend.

Tip: Use `static_cast` for safe conversions between numeric types. Use `dynamic_cast` for safe downcasting in class hierarchies. Avoid `const_cast` and `reinterpret_cast` unless you have a very specific reason. Never use C-style casts in new C++ code.

Time

Programs often need to work with time — measuring how long something takes, pausing execution, or converting between time units. C++ provides the `<chrono>` library for this.

Measuring Elapsed Time

The most common use of `<chrono>` is measuring how long a piece of code takes to run. `std::chrono::steady_clock` is the right clock for this because it never jumps forward or backward. The key functions are:

```
static time_point steady_clock::now();          // current time
Duration duration_cast<Duration>(duration d);   // convert between time units
Rep duration::count() const;                    // get the numeric value
void this_thread::sleep_for(duration d);        // pause execution
```

```
#include <chrono>
#include <iostream>
#include <thread>

int main()
{
    auto start = std::chrono::steady_clock::now();

    // Simulate some work
    std::this_thread::sleep_for(std::chrono::milliseconds(150));

    auto end = std::chrono::steady_clock::now();
    auto elapsed =
        std::chrono::duration_cast<std::chrono::milliseconds>(
            end - start);

    std::cout << "That took " << elapsed.count() << " ms" << std::endl;

    return 0;
}
```

Output (approximately):

```
That took 150 ms
```

`steady_clock::now()` returns a time point. Subtracting two time points gives a duration. `duration_cast` converts that duration to the units you want — milliseconds, microseconds, seconds, etc.

Duration Arithmetic

Durations are type-safe. You cannot accidentally mix up seconds and milliseconds because they are different types. The library handles conversions automatically when it is safe.

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::chrono;

    seconds two_min = minutes(2);
    milliseconds half_sec = milliseconds(500);

    std::cout << "2 minutes = "
              << two_min.count()
              << " seconds" << std::endl;
    std::cout << "500 ms = " << duration_cast<seconds>(half_sec).count()
              << " seconds" << std::endl;

    auto mixed = seconds(3) + milliseconds(250);
    std::cout << "3s + 250ms = "
              << duration_cast<milliseconds>(mixed).count()
              << " ms" << std::endl;

    return 0;
}
```

Output:

```
2 minutes = 120 seconds
500 ms = 0 seconds
3s + 250ms = 3250 ms
```

Notice that converting 500 milliseconds to seconds gives `0`, not `0.5`. `duration_cast` truncates — it does not round. This is the same behavior as integer division.

Tip: Use `std::chrono::steady_clock` for measuring elapsed time. `system_clock` can jump forward or backward (e.g., when the system clock is adjusted), which would throw off your measurements.

Trap: `duration_cast` truncates toward zero. If you need to know that an operation took 1.7 seconds, cast to milliseconds (1700) rather than seconds (1).

Random Numbers

Generating random numbers comes up surprisingly often — games, simulations, testing, shuffling data. C++ provides a proper random number library in `<random>`.

The Old Way: rand()

C provides `rand()` and `srand()` in `<cstdlib>`. Their signatures are:

```
int rand();                          // returns a pseudo-random integer
void srand(unsigned int seed);       // seeds the random number generator
// returns current calendar time (from <ctime>)
time_t time(time_t* arg);
```

You might see them in older code:

```
#include <cstdlib>
#include <ctime>
#include <iostream>

int main()
{
    srand(static_cast<unsigned>(time(nullptr)));  // Seed with current time
    std::cout << rand() % 10 << std::endl;        // 0-9, but biased!

    return 0;
}
```

This works but has problems. `rand()` produces low-quality random numbers on many systems. Using `%` to get a range introduces bias — some numbers come up more often than others. And `srand(time(nullptr))` means two programs started in the same second get the same sequence.

Trap: Avoid `rand()` and `srand()` in new C++ code. They exist for C compatibility but produce poor randomness and make it easy to introduce subtle bias.

The C++ Way: Engines and Distributions

The `<random>` library separates two concerns: generating raw random bits (the **engine**) and shaping those bits into the range and distribution you want (the **distribution**). The key components and their signatures are:

```
// std::random_device
unsigned int operator()();   // produces a random seed

// std::mt19937
mt19937(unsigned int seed);  // construct with seed

// distributions
uniform_int_distribution(IntType a, IntType b);     // integers in [a, b]
uniform_real_distribution(RealType a, RealType b);   // reals in [a, b)
ResultType operator()(Generator& gen);               // generate a value
```

```
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<int> track(1, 12);
```

```
    std::uniform_real_distribution<double> score(0.0, 10.0);

    std::cout << "Random track: " << track(gen) << std::endl;
    std::cout << "Random score: " << score(gen) << std::endl;

    std::cout << "Cinco tracks al azar: ";
    for (int i = 0; i < 5; ++i) {
        std::cout << track(gen) << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Possible output:

```
Random track: 7
Random score: 3.14159
Cinco tracks al azar: 11 3 7 1 9
```

Here is what each piece does:

- **`std::random_device rd`** provides a seed from your operating system's entropy source — truly unpredictable.
- **`std::mt19937 gen(rd())`** creates a Mersenne Twister engine seeded with that random value. This engine produces high-quality pseudo-random numbers.
- **`std::uniform_int_distribution<int> track(1, 12)`** takes the engine's output and maps it to an integer in $[1, 12]$, with each value equally likely.
- **`std::uniform_real_distribution<double> score(0.0, 10.0)`** does the same for floating-point values in $[0.0, 10.0)$.

Tip: Create the engine once and reuse it. Creating a new `std::mt19937` for every random number is wasteful and can produce poor results if seeded with similar values.

Wut: `std::random_device` is not guaranteed to be truly random on all platforms. On some systems it may fall back to a pseudo-random generator. In practice, on Linux, macOS, and Windows, it reads from the OS entropy pool and is fine for seeding.

Other Distributions

`uniform_int_distribution` and `uniform_real_distribution` give every value in the range an equal chance. But sometimes you want values that cluster around a center — this is a **normal distribution** (also called a Gaussian or bell curve).

```
std::normal_distribution<RealType>(RealType mean, RealType stddev);
```

The `mean` is the center of the bell curve. The `stddev` (standard deviation) controls how spread out the values are — about 68% of values fall within one standard deviation of the mean, and about 95% within two.

```
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;
```

```
    std::mt19937 gen(rd());

    std::normal_distribution<double> rating(7.0, 1.5);

    std::cout << "Diez puntuaciones al azar:" << std::endl;
    for (int i = 0; i < 10; ++i) {
        std::cout << rating(gen) << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Most values will be close to 7.0, with occasional values farther away. The `<random>` header provides many other distributions (Bernoulli, Poisson, etc.), but uniform and normal cover most practical needs.

Key Points

- `exit()` terminates the program from any function; prefer `return` from `main()` when possible.
- `EXIT_SUCCESS` and `EXIT_FAILURE` are portable constants for exit codes.
- C++ mangles function names to support overloading; C does not.
- `extern "C"` disables name mangling so C and C++ code can link together.
- Use `#ifdef __cplusplus` guards to write headers that work in both C and C++.
- To the CPU, everything is a number — types tell the compiler how to interpret the bits.
- A `char` is just a small integer; `'A'` is 65.
- Different bit widths give different value ranges; overflow wraps around.
- Prefer `static_cast` for safe conversions, `dynamic_cast` for safe downcasting, and avoid `const_cast` and `reinterpret_cast` unless necessary.
- Never use C-style casts in new C++ code — use the named C++ casts instead.
- Use `std::chrono::steady_clock` to measure elapsed time; `duration_cast` converts between time units but truncates.
- Avoid `rand()` and `srand()` — use `<random>` with an engine (`std::mt19937`) and a distribution (`std::uniform_int_distribution`, etc.).
- Seed the engine with `std::random_device` for unpredictable results.
- `std::normal_distribution` generates values clustered around a mean with a given standard deviation (bell curve).

Exercises

1. What does the following program print if the file `data.txt` does not exist?

```
#include <cstdlib>
#include <fstream>
#include <iostream>

void read_file()
{
    std::ifstream f("data.txt");
    if (!f) {
        std::cout << "A" << std::endl;
        exit(EXIT_FAILURE);
    }
    std::cout << "B" << std::endl;
}
```

```
int main()
{
    read_file();
    std::cout << "C" << std::endl;
    return EXIT_SUCCESS;
}
```

2. What is name mangling, and why does C++ do it but C does not?
3. A coworker writes the following C++ code to call a C library function but gets a linker error about an undefined symbol. What is the fix?

```
// my_program.cpp
#include <iostream>

void c_library_init();

int main()
{
    c_library_init();
    std::cout << "Ready" << std::endl;
    return 0;
}
```

4. What is the value of `x` after this code runs?

```
uint8_t x = 250;
x = x + 10;
```

5. What does the following program print?

```
#include <iostream>

int main()
{
    char c = 48;
    std::cout << c << std::endl;
    std::cout << static_cast<int>(c) << std::endl;
    return 0;
}
```

6. Explain why this C-style cast is dangerous and what C++ cast you should use instead:

```
void* ptr = get_some_pointer();
int* ip = (int*)ptr;
```

7. What is the difference between `static_cast<int>(3.14)` and `reinterpret_cast<int>(3.14)`? Will the second one even compile?
8. What does the `#ifdef __cplusplus` guard accomplish in a C/C++ shared header? When would the code inside the `#ifdef` be skipped?
9. Write a program that takes an `int` and prints it as a `char`, and takes a `char` and prints its integer value. Use `static_cast` for both conversions. Test it with the value `65` and the character `'Z'`.
10. What does the following program print?

```
#include <chrono>
#include <iostream>

int main()
```

```
{
    using namespace std::chrono;

    auto d = seconds(5) + milliseconds(750);
    std::cout << duration_cast<seconds>(d).count() << std::endl;

    return 0;
}
```

11. Why should you use `std::chrono::steady_clock` instead of `std::chrono::system_clock` when measuring how long a piece of code takes to run?

12. What is wrong with this code for generating a random number between 1 and 100?

```
#include <cstdlib>
#include <iostream>

int main()
{
    int r = rand() % 100 + 1;
    std::cout << r << std::endl;

    return 0;
}
```

13. Write a program that uses `<random>` to simulate rolling two six-sided dice 10 times and prints each roll.

14. Write a program that generates 10 random values using `std::normal_distribution` with a mean of 100 and a standard deviation of 15. Print each value. Are most values close to 100?

15. **What does this print?**

```
#include <iostream>

struct Track { virtual ~Track() = default; };
struct AudioTrack : Track { void play() { std::cout << "audio\n"; } };
struct VideoTrack : Track { void play() { std::cout << "video\n"; } };

void play(Track *t)
{
    if (auto *a = dynamic_cast<AudioTrack *>(t)) {
        a->play();
    } else if (auto *v = dynamic_cast<VideoTrack *>(t)) {
        v->play();
    } else {
        std::cout << "unknown\n";
    }
}

int main()
{
    AudioTrack a;
    VideoTrack v;
    Track      t;
    play(&a);
    play(&v);
    play(&t);
```

```
    return 0;
}
```

Why does the base class need a `virtual` destructor for `dynamic_cast` to work here?

16. **Where is the bug?**

```
void uppercase_first(const std::string &s)
{
    char &first = const_cast<char &>(s[0]);
    first = static_cast<char>(std::toupper(first));
}

int main()
{
    const std::string title = "wonderwall";
    uppercase_first(title);
    std::cout << title << "\n";
    return 0;
}
```

The function compiles, but it is undefined behavior at runtime. Explain what `const_cast` is doing here and why this particular use of it is broken.

17. **Calculation:** Given the fixed-width types from `<cstdint>`, how many bytes does each of the following take, and what is the largest value it can hold?

```
int8_t   a;
uint8_t  b;
int16_t  c;
uint32_t d;
int64_t  e;
```

Why prefer `int32_t` over `int` when you need exactly 32 bits, and why prefer `int` over `int32_t` for ordinary counters?

18. **Think about it:** What is the difference between calling `std::exit(0)` and writing `return 0;` from `main`? Specifically, which destructors run in each case? Sketch a small program with a class that prints from its destructor and predict what each version prints.

19. **Write a program** that seeds a `std::mt19937` from `std::random_device`, then uses it with a `std::uniform_int_distribution<int>(1, 100)` to print 10 random integers in the range `[1, 100]`. Now run the program twice and compare: do you get the same numbers each time? Then change the program to use a fixed seed (`std::mt19937 rng(42);`) and run it twice again. What changed, and why?

References

[1] B. W. Kernighan and D. M. Ritchie, *The C programming language*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1988.

Index