# Gorgo Starting C++ — Answer Key

April 11, 2026

## Contents



## Chapter 1: Introduction

**1. What does the following program print?**

```cpp
#include <iostream>

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

It prints:

```
AB
C
```

The first line chains `"A"` and `"B"` together on the same line, then `std::endl` ends the line. The second line prints `"C"` on a new line.

**2. What is wrong with the following program?**

```cpp
#include <iostream>

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

There is a missing semicolon at the end of the `std::cout` line. The line `std::cout << "Here we are now" << std::endl` needs a `;` after `std::endl`. Without it, the compiler sees `std::endl return` which is not valid C++.

**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?**

`std::cout` lives in the `std` namespace, which is where the C++ standard library places all of its names. The `std::` prefix tells the compiler to look for `cout` inside the `std` namespace. If you removed `std::` without adding `using namespace std;`, the compiler would not know where to find `cout` and would report an error saying `cout` was not declared.

**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?**

The `-o hello` flag tells the compiler to name the output executable `hello`. If you left it out, the compiler would use the default output name, which is typically `a.out` on Linux and macOS.

**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`?** It prints `beta`. `argv[0]` is `./program`, `argv[1]` is `alpha`, `argv[2]` is `beta`, and `argv[3]` is `gamma`.

**What happens if you run it with `./program alpha`?** This is undefined behavior. `argc` is 2, so `argv[2]` is out of bounds. The program might crash, print garbage, or do something else unpredictable.

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

3 arguments. `argc` counts all arguments including the program name (`argv[0]`), so the user provided `argc - 1 = 3` arguments.

**7. Write a program that asks for the user's name and favorite number, then prints a message using both.**

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

int main()
{
    std::string name;
    int number;

    std::cout << "What is your name? ";
    std::getline(std::cin, name);

    std::cout << "What is your favorite number? ";
    std::cin >> number;

    std::cout << "Hola, " << name << "! Your favorite number is "
              << number << "." << std::endl;

    return 0;
}
```

**8. Think about it: What is the difference between `std::endl` and `"\n"`?**

Both write a newline character, so the cursor moves to the next line in either case. The difference is that `std::endl` *also* flushes the output buffer. `"\n"` is just a character — the runtime is free to keep it (and everything before it) sitting in the buffer until the buffer fills up or the program flushes it some other way (program exit, `std::cin >>`, an explicit `std::cout.flush()`, etc.).

When does the difference matter? Anywhere you need the user to see output *right now*, before the next thing happens: progress messages during a long computation, prompts that come before a `std::cin`, log lines you want to read while a program is still running, and so on. For most everyday output the difference is invisible because the buffer is flushed often enough on its own, and `"\n"` is faster (no flush).

**9. Where is the bug? "Loading..." does not appear until the program exits.**

`std::cout` is buffered. `"Loading..."` has no newline at the end and nothing in the program flushes the buffer, so the characters sit in `std::cout`'s buffer through the long computation and only get written when the program exits and the buffer is finally flushed.

The fix is to flush the buffer explicitly so the message reaches the terminal before the computation runs:

```cpp
std::cout << "Loading..." << std::endl;
// or
std::cout << "Loading..." << std::flush;
```

Either change makes "Loading…" appear immediately.

**10. Reading "Como estas" with `std::cin >>` vs `std::getline`.**

With `std::cin >> greeting;` the program prints `[Como]`. The `>>` operator on `std::string` reads characters until it sees whitespace, so it stops at the space between `Como` and `estas` and only the first word ends up in `greeting`. The rest of the line (`estas\n`) is left sitting in the input buffer.

With `std::getline(std::cin, greeting);` the program prints `[Como estas]`. `std::getline` reads characters until it sees a newline (which it consumes but does not store), so the entire `Como estas` becomes one string, spaces and all.

Use `>>` when you want to read one whitespace-delimited token; use `std::getline` when you want a whole line of input including any internal spaces.

**11. Where is the bug? `"He said "wassup" and left."` will not compile.**

The compiler reads a string literal as everything between an opening `"` and the next `"`. Given `"He said "wassup" and left."` it sees the string `"He said "`, then a stray identifier `wassup`, then another string `" and left."`, and gets confused. The inner double quotes need to be escaped so they become part of the string instead of ending it:

```
std::cout << "He said \"wassup\" and left." << std::endl;
```

That prints `He said "wassup" and left.` as intended.

**12. What does `std::cout << "a\\b\tc\nd" << std::endl;` print?**

```
a\b c
d
```

`\\` is a single backslash, `\t` is a tab, and `\n` is a newline, so the string is six characters: `a`, `\`, `b`, tab, `c`, newline, `d`, and then `std::endl` adds another newline.

# Chapter 2: Variables

**1. On a system where `int` is 4 bytes, what is `sizeof(scores)` for `int scores[10]`?**

40 bytes. Each `int` is 4 bytes, and the array has 10 elements, so `sizeof(scores)` is `4 * 10 = 40`.

**2. What does the following program print?**

```
#include <iostream>

int main()
{
    char c = 'C';
    c = c + 3;
    std::cout << c << std::endl;
    return 0;
}
```

It prints:

```
F
```

`'C'` has ASCII value 67. Adding 3 gives 70, which is the ASCII value of `'F'`.

**3. What is wrong with the following code?**

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

The array `data` has 3 elements with valid indices 0, 1, and 2. `data[3]` is an out-of-bounds access, which is undefined behavior. The last valid element is `data[2]`.

**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?** `const int *p1` prevents you from changing the value being pointed to. You cannot write `*p1 = 42`.

**Which one prevents you from changing where the pointer points?** `int *const p2` prevents you from changing where the pointer points. You cannot write `p2 = &other_variable`. Note that `p2` must be initialized when declared because it is a `const` pointer — it can never be reassigned.

**5. What does the following program print?**

```cpp
#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;
}
```

It prints:

```
3 10
```

When `b = a` is executed, all members of `a` are copied into `b`. Modifying `b.x` does not affect `a.x` because `b` has its own copy of the data.

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

An uninitialized variable contains whatever garbage data was previously in that memory location. Reading from an uninitialized `int` is undefined behavior. The value could be anything — zero, a large number, a negative number — and it may be different each time you run the program. This makes bugs extremely hard to track down because the program may appear to work sometimes and fail other times.

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

An `unsigned short` with 2 bytes (16 bits) can hold values from 0 to 65,535 (2^16 - 1). A signed `short` with 2 bytes can hold values from -32,768 to 32,767 (-2^15 to 2^15 - 1). The signed version uses one bit for the sign, which halves the positive range but allows negative values.

**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.**

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

struct Car {
    std::string make;
    std::string model;
    int year;
};

int main()
{
    Car cars[3] = {
        {"Honda", "Civic", 1995},
        {"Toyota", "Corolla", 1998},
        {"Ford", "Mustang", 1994}
    };
```

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

    for (int i = 0; i < count; i++) {
        std::cout << cars[i].year << " " << cars[i].make
                  << " " << cars[i].model << std::endl;
    }

    return 0;
}
```

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

`std::numeric_limits<uint8_t>::max()` returns `255` — the largest value an 8-bit unsigned integer can hold.

`std::numeric_limits<double>::min()` is *not* a large negative number. It returns the smallest *positive* normalized `double` value (approximately 2.2e-308). To get the most negative `double`, use `std::numeric_limits<double>::lowest()`.

**10. What does the `auto` example print, and what types are deduced?**

It prints `42 42 8 8.4`.

- `auto a = 42;` deduces `int` (the literal `42` is `int`).
- `auto b = 42.0;` deduces `double` (the literal `42.0` is `double`).
- `auto c = 42 / 5;` deduces `int`. Both operands are `int`, so this is integer division: 42 / 5 == 8.
- `auto d = 42.0 / 5;` deduces `double`. One operand is `double`, so the other is converted and the division is floating point: 42.0 / 5 == 8.4.

`auto` is convenient, but you have to know the rules of the right-hand side to predict the type.

**11. Calculation: `grid[1][2]`, `sizeof(grid)`, total elements.**

- `grid[1][2]` is `6`. `grid[1]` is the second row `{4, 5, 6, 7}`, and index `2` of that row is `6`.
- `sizeof(grid)` is `48` bytes. The grid has 3 * 4 == 12 `int` elements, and `int` is 4 bytes, so 12 * 4 == 48.
- The grid holds 12 `int` elements total.

**12. What does the `unsigned char x = 250; x = x + 10;` example print?**

It prints `4`.

`x` is an 8-bit `unsigned char`, so it can hold values from 0 to 255. The arithmetic `x + 10` would be 260, which does not fit in 8 bits. For unsigned types this is **wraparound**: the value goes off the top end and reappears at 0, so 260 becomes 260 - 256 = 4. This is well-defined behavior for unsigned types — the standard guarantees the result is taken modulo 2^N. (Signed integer overflow is **undefined** behavior, which you will learn more about in Chapter 7.)

The `static_cast<int>(x)` is just so `std::cout` prints `x` as a number instead of as a character; without it, `x` would be printed as the unprintable character with code 4.

**13. What does the `char a = 'a'; char b = a + 4;` example print?**

It prints `e 101`.

From the ASCII table, `'a'` is 97. `a + 4` is 101, which the ASCII table maps to `'e'`. When `b` is sent to `std::cout` as a `char`, it is displayed as the glyph `e`. When `static_cast<int>(b)` is sent, it is displayed as the number `101`.

Same byte, two different displays — the type controls which one you see.

# Chapter 3: Strings

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

`std::cin >> str` reads one word at a time, stopping at the first whitespace character (space, tab, or newline). `std::getline(std::cin, str)` reads an entire line of input, including spaces, until it hits a newline.

Use `std::cin >>` when you want to read a single word or token. Use `std::getline()` when you need to read input that may contain spaces, such as a full name or a sentence.

**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;
```

It prints:

```
Ice Ice Baby
12
```

The string `b` is built by concatenating `"Ice"`, `" "`, `"Ice"`, and `" Baby"`, producing `"Ice Ice Baby"` which has 12 characters.

**3. What is `std::string("Hola").at(4)`? What about `std::string("Hola")[4]`?**

`std::string("Hola").at(4)` throws a `std::out_of_range` exception. The string `"Hola"` has indices 0 through 3, so index 4 is out of bounds and `.at()` catches this.

`std::string("Hola")[4]` accesses the null terminator character `'\0'`. The `[]` operator does not perform bounds checking, and `std::string` stores a null terminator at position `size()`, so `[4]` returns `'\0'`.

**4. What is the value of `pos` after this code runs?**

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

`pos` is 15. The substring `"dop"` starts at index 15 in the string `"MMMBop ba duba dop"`.

**5. Where is the bug in this code?**

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

You cannot concatenate two string literals with `+`. Both `"Hello, "` and `"world!"` are C-style string literals (character arrays), not `std::string` objects. At least one side of `+` must be a `std::string`. Fix it by making one side a `std::string`:

```
std::string greeting = std::string("Hello, ") + "world!";
```

**6. Where is the bug in this program?**

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

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;
}
```

After `std::cin >> count` reads the integer, the newline character from pressing Enter is left in the input buffer. The subsequent `std::getline()` sees that leftover newline and immediately returns an empty string without waiting for user input. Fix it by adding `std::cin.ignore()` between the `>>` and `getline()` calls:

```
std::cin >> count;
std::cin.ignore();
std::getline(std::cin, name);
```

**7. What does this code print?**

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

It prints:

```
B@il@mos
```

The loop replaces every lowercase `'a'` with `'@'`. The `'B'` is uppercase and not affected.

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

`std::stoi("abc42")` throws a `std::invalid_argument` exception. `std::stoi` starts parsing from the beginning of the string. `"42abc"` starts with valid digits so it parses 42 and stops at `'a'`. `"abc42"` starts with non-digit characters so there is nothing valid to parse, and it throws an exception.

**9. Write a program that asks the user for their full name using `std::getline()`, then prints the number of characters in their name and their name in reverse.**

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

int main()
{
    std::string name;

    std::cout << "Enter your full name: ";
    std::getline(std::cin, name);

    std::cout << "Your name has " << name.size() << " characters." << std::endl;

    std::cout << "Reversed: ";
    for (int i = static_cast<int>(name.size()) - 1; i >= 0; i--) {
        std::cout << name[i];
    }
    std::cout << std::endl;
```

```
    return 0;
}
```

**10. What does the `lyric.replace(0, 3, "Pop")` example print?**

It prints:

```
Pop bop, ba duba dop
```

`replace(pos, len, str)` removes `len` characters starting at index `pos` and inserts `str` in their place. Here it removes the 3 characters `"Mmm"` starting at index 0 and inserts `"Pop"`, leaving the rest of the string unchanged. Note that the inserted string does not have to be the same length as the removed range.

**11. What does the `substr` example print?**

It prints:

```
Wann
nabe
nabe
```

- `title.substr(0, 4)` extracts 4 characters starting at index 0: `"Wann"`.
- `title.substr(3)` (no length argument) extracts everything from index 3 to the end of the string: `"nabe"`.
- `title.substr(3, 100)` asks for 100 characters starting at index 3, but there are only 4 characters left in the string. `substr` does not crash or throw — it silently clamps the requested length to whatever is available, so you get `"nabe"` again.

(If the *starting* index were past the end of the string, `substr` would throw `std::out_of_range`. Only an over-long *length* is silently clamped.)

**12. Think about it: case-sensitive comparison `"Wonderwall"` vs `"wonderwall"`.**

It prints:

```
0
1
```

`std::string` comparison compares characters by their numeric (ASCII) values. `'W'` is ASCII 87 and `'w'` is ASCII 119, so the very first character of `a` is *less than* the first character of `b`, which makes the entire string `a` less than `b`. Equality (`==`) returns `0` (false) because the first characters differ; ordering (`<`) returns `1` (true) because uppercase letters come before lowercase ones in ASCII.

To compare case-insensitively you have to normalize the case yourself first — for example, copy both strings, convert each character with `std::tolower` from `<cctype>`, then compare:

```
#include <cctype>

std::string lower(std::string s)
{
    for (char &c : s) c = static_cast<char>(std::tolower(static_cast<unsigned char>(c)));
    return s;
}

bool equal_ignore_case(const std::string &x, const std::string &y)
{
    return lower(x) == lower(y);
}
```

`std::string` has no built-in case-insensitive compare; you build it yourself like this.

**13. What does `std::string s = "café"; s.size();` report?**

It prints:

```
café 5
```

The string has 4 visible characters but takes 5 bytes in UTF-8. `c`, `a`, and `f` are ASCII and take one byte each (3 bytes). `é` is Unicode code point U+00E9, which UTF-8 encodes as the two bytes `0xC3 0xA9` (2 bytes). $3 + 2 = 5$.

`std::string::size()` always reports bytes, not characters. For a pure-ASCII string the two would be the same, but as soon as a non-ASCII character shows up, the byte count exceeds the human "character" count.

# Chapter 4: Expressions

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

`7 / 2` performs integer division and produces `3`. The fractional part is discarded because both operands are integers.

`7.0 / 2` performs floating-point division and produces `3.5`. Because at least one operand is a floating-point type (`7.0` is a `double`), the other operand is promoted to `double` before the division.

This matters because integer division silently drops the decimal part, which can lead to incorrect results if you expect a fractional answer.

**2. What does the following code print?**

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

It prints:

```
12 10 12
```

- `b = a++`: postfix returns the current value of `a` (10) and then increments `a` to 11. So `b` is 10, `a` is 11.
- `c = ++a`: prefix increments `a` to 12 first, then returns the new value. So `c` is 12, `a` is 12.

**3. What is the value of each expression?**

- `17 % 5` = **2** ($17 = 3 * 5 + 2$)
- `20 % 4` = **0** ($20 = 5 * 4 + 0$)
- `3 % 7` = **3** ($3 = 0 * 7 + 3$, since 3 is less than 7)

**4. What does this expression evaluate to?**

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

`result` is `false`.

It does not crash because of short-circuit evaluation. The left side `(x != 0)` evaluates to `false`. Since `&&` requires both sides to be true and the left side is already `false`, the right side `(100 / x > 5)` is never evaluated. The division by zero never happens.

**5. Where is the bug?**

```
int x = 5;
if (x = 10) {
    std::cout << "x is 10" << std::endl;
}
```

The condition uses `=` (assignment) instead of `==` (comparison). `x = 10` assigns 10 to `x` and then the expression evaluates to 10, which is non-zero and therefore `true`. The `if` block always executes regardless of `x`'s original value. The fix is to use `==`:

```
if (x == 10) {
```

**6. Where is the bug?**

```
int flags = 10;
if (flags & 2 == 2) {
    std::cout << "bit is set" << std::endl;
}
```

The `==` operator has higher precedence than `&`. So the expression is parsed as `flags & (2 == 2)`, which evaluates to `flags & 1`, not `(flags & 2) == 2`. The fix is to add parentheses:

```
if ((flags & 2) == 2) {
```

**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;
```

It prints:

```
B
```

`score` is 85. The first condition `score >= 90` is false, so it moves to the next. The second condition `score >= 80` is true, so `grade` is set to `"B"`.

**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).**

```
#include <iostream>

int main()
{
    int n;

    std::cout << "Enter an integer: ";
    std::cin >> n;

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

    if (n > 0) {
        std::cout << n << " is positive" << std::endl;
    } else if (n < 0) {
        std::cout << n << " is negative" << std::endl;
    } else {
        std::cout << n << " is zero" << std::endl;
    }
```

```
    return 0;
}
```

**9. What does the compound-assignment example print?**

It prints `0`.

| Line | New value of `x` |
|---|---|
| `x = 10` | 10 |
| `x += 5` | 15 |
| `x *= 2` | 30 |
| `x -= 3` | 27 |
| `x /= 4` | 6 (integer div) |
| `x %= 5` | 1 |

Wait — 6 % 5 is 1, not 0. Trace it again carefully: 10 + 5 = 15, 15 * 2 = 30, 30 - 3 = 27, 27 / 4 = 6 (integer division drops the remainder), 6 % 5 = 1. The program prints `1`.

**10. Think about it: precedence of `a < b && c == d || !e`.**

Operator precedence in this expression, from highest to lowest:

1. `!e` (unary NOT) is evaluated first.
2. `a < b` and `c == d` (relational and equality) are evaluated next.
3. `a < b && c == d` (logical AND) binds more tightly than `||`.
4. `... || !e` (logical OR) is evaluated last.

So C++ reads it as `((a < b) && (c == d)) || (!e)`.

The explicit version is preferable because it costs nothing to read (no precedence rules to recall) and it removes any temptation to "fix" it later by reordering operators. You may remember the precedence rules today; the next reader of the code (including future-you) might not. The CLAUDE.md style guide for this book even has a tip recommending parentheses whenever you mix logical operators for exactly this reason.

# Chapter 5: Control Flow

**1. Think about it: When would you choose a `do-while` loop over a `while` loop?**

You would choose a `do-while` loop when the loop body must execute at least once before the condition is tested. A classic example is an input validation loop where you want to ask the user for input and then check if it is valid. With a `while` loop you would have to duplicate the prompt before the loop to set up the first test, but a `do-while` handles this naturally. Another example is a menu system: you always want to display the menu at least once before checking if the user chose to quit.

**2. What does this print?**

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

It prints:

```
0 1 2 4
```

The loop iterates from 0 to 4. When `i` is 3, `continue` skips the rest of the loop body (the `std::cout`), so 3 is not printed.

**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";
```

It prints:

```
dos tres
```

`x` is 2, so execution jumps to `case 2`. There is no `break` after `case 2`, so execution falls through into `case 3`, printing `"tres "`. The `break` in `case 3` stops the fall-through.

**4. Where is the bug?**

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

There is a stray semicolon at the end of the `for` line: `for (i = 0; i < 10; i++);`. The semicolon makes the `for` loop's body an empty statement, so the loop runs 10 times doing nothing. The block `{ total += i; }` is a separate block that runs once after the loop finishes, when `i` is 10. The program prints `Total: 10` instead of the intended `Total: 45`. The fix is to remove the semicolon after the `for` statement.

**5. Calculation: How many times does the body of this loop execute?**

```
int count = 0;
int i = 10;
do {
    count++;
    i--;
} while (i > 10);
```

The body executes **1 time**. A `do-while` loop always executes the body at least once before testing the condition. After the first iteration, `i` is 9, and the condition `i > 10` is false, so the loop stops. `count` is 1.

**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";
```

It prints:

```
tres cinco tres tres cinco tres both tres cinco
```

The numbers from 1 to 20 that are divisible by 3 or 5:

- 3: tres
- 5: cinco
- 6: tres
- 9: tres
- 10: cinco
- 12: tres
- 15: both (divisible by both 3 and 5)
- 18: tres
- 20: cinco

Numbers not divisible by 3 or 5 produce no output.

**7. Where is the bug?**

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

`n` starts at 0 and increases by 3 each iteration: 0, 3, 6, 9, 12, 15, … The value 10 is never reached, so the condition `n != 10` is always true and the loop runs forever. The fix is to use `<` instead of `!=`:

```
while (n < 10) {
```

This makes the loop safe even if `n` skips over the exact target.

**8. Write a program that asks the user for a number between 1 and 7, prints the day of the week, and uses a `do-while` loop to keep asking until the user enters 0 to quit.**

```
#include <iostream>

int main()
{
    int choice;

    do {
        std::cout << "Enter a day (1-7, 0 to quit): ";
        std::cin >> choice;

        switch (choice) {
        case 1:
            std::cout << "Monday" << std::endl;
            break;
        case 2:
            std::cout << "Tuesday" << std::endl;
            break;
        case 3:
```

```
            std::cout << "Wednesday" << std::endl;
            break;
        case 4:
            std::cout << "Thursday" << std::endl;
            break;
        case 5:
            std::cout << "Friday" << std::endl;
            break;
        case 6:
            std::cout << "Saturday" << std::endl;
            break;
        case 7:
            std::cout << "Sunday" << std::endl;
            break;
        case 0:
            std::cout << "Adios!" << std::endl;
            break;
        default:
            std::cout << "Invalid number. Try 1-7 or 0 to quit." << std::endl;
            break;
        }
    } while (choice != 0);

    return 0;
}
```

**9. What does the nested-loop triangle program print?**

The first version prints

```
*
**
***
```

The outer loop runs `row` from 1 to 3 and the inner loop prints `row` stars on each line.

Replacing the inner loop with `for (int col = 1; col <= 4 - row; ++col)` flips the pattern, because now each row prints `4 - row` stars: 3 on the first row, 2 on the second, 1 on the third.

```
***
**
*
```

**10. What does the range-based `for` loop print, and why use `const std::string &`?**

It prints:

```
- Wonderwall
- Creep
- Linger
```

Using `const std::string &` instead of plain `std::string` avoids copying each string for every iteration — the loop binds the reference directly to the element in the vector. The `const` makes it clear that the loop body will not modify the elements (and lets the compiler help enforce that). For a tiny type like `int` the copy is free and you can just write `int x : numbers`, but for `std::string`, large structs, or anything that owns memory, prefer `const &`.

**11. Write a program that uses `break` to find the first negative number in an array.**

```cpp
#include <iostream>

int main()
{
    int values[] = {3, 7, 2, -5, 4, -1};
    int size = sizeof(values) / sizeof(values[0]);

    int found_index = -1;
    for (int i = 0; i < size; ++i) {
        if (values[i] < 0) {
            found_index = i;
            break;
        }
    }

    if (found_index >= 0) {
        std::cout << "first negative is " << values[found_index]
                  << " at index " << found_index << "\n";
    } else {
        std::cout << "none\n";
    }

    return 0;
}
```

`break` exits the `for` loop the moment we find a negative element, so we do not waste time scanning the rest of the array. The sentinel `found_index = -1` lets us tell "found nothing" apart from "found something at index 0" after the loop.

**12. Think about it: intentional `switch` fall-through.**

Fall-through is useful when several cases should run the same code. A common example is grouping characters that should be treated identically:

```cpp
switch (c) {
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
    std::cout << "vowel\n";
    break;
default:
    std::cout << "consonant\n";
    break;
}
```

Each empty `case` falls through into the next one, so any vowel ends up running the same body. This pattern is so idiomatic that no annotation is needed; the empty cases make the intent obvious.

When the fall-through is between *non-empty* cases, the C++17 attribute `[[fallthrough]]` tells both the compiler and human readers that the missing `break` is intentional:

```cpp
switch (mode) {
case 1:
    setup();
    [[fallthrough]];
```

```
case 2:
    run();
    break;
}
```

Without `[[fallthrough]]`, modern compilers warn about the missing `break` because that is almost always a bug.

# Chapter 6: Functions

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

Pass-by-value gives you a guarantee that the function cannot modify the caller's variable. When you pass by value, the function gets its own copy, so you can reason about your code locally — you know that calling a function will not change your variables unexpectedly. This makes code easier to understand and debug because you do not need to look inside a function to know whether it modifies its arguments.

**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;
}
```

It prints:

```
5 15
```

`a` is passed by value, so modifying it inside `mystery` does not affect `x`. `b` is passed by reference, so adding 10 to `b` modifies `y` directly.

**3. Calculation: What does `factorial(6)` return?**

`factorial(6)` returns **720**.

6! = 6 * 5 * 4 * 3 * 2 * 1 = 720.

**4. Where is the bug?**

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

There is no base case. The function calls itself forever (with decreasing values of `n` passing through 0 and into negative numbers) until the stack overflows and the program crashes. The fix is to add a base case:

```
int countdown(int n) {
    if (n <= 0) {
        return 0;
    }
    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;
}
```

It prints:

```
Hola, Mack
Hola, Mack! Hola, Mack! Hola, Mack!
```

The first call matches the one-parameter overload. The second call matches the two-parameter overload, which prints the greeting 3 times on one line.

**6. Where is the bug?**

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

Default parameters must appear at the end of the parameter list. Here, `volume` has a default value but `song` (which comes after it) does not. This is a compilation error. The fix is to reorder the parameters:

```
void set_volume(const std::string &song, int volume = 5) {
    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;
}
```

It prints:

```
7
12
```

`apply(add, 3, 4)` calls `add(3, 4)` which returns $3 + 4 = 7$. `apply(mul, 3, 4)` calls `mul(3, 4)` which returns $3 * 4 = 12$.

**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;
}
```

It prints:

```
11
0
```

`a + b` calls `operator+`, which adds the levels: $5 + 6 = 11$. `a > b` calls `operator>`, which compares $5 > 6$, which is false (0).

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

The built-in `&&` and `||` use **short-circuit evaluation**: the right operand is only evaluated if the left operand does not already determine the result. For example, `ptr != nullptr && ptr->valid()` is safe because if `ptr` is null, the right side is never evaluated.

When you overload `&&` or `||`, both operands are always evaluated before the operator function is called, because function arguments are evaluated before the function runs. This means `ptr != nullptr && ptr->valid()` would crash if `ptr` is null, because `ptr->valid()` would be evaluated regardless.

**10. Write a program with `is_even` and `count_if` functions.**

```
#include <iostream>

bool is_even(int n) {
    return n % 2 == 0;
}

int count_if(const int arr[], int size, bool (*predicate)(int)) {
    int count = 0;
    for (int i = 0; i < size; i++) {
        if (predicate(arr[i])) {
            count++;
        }
```

```
    }
    return count;
}

int main() {
    int numbers[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    int evens = count_if(numbers, size, is_even);
    std::cout << "Even numbers: " << evens << std::endl;

    return 0;
}
```

This prints `Even numbers: 5` because there are 5 even numbers (2, 4, 6, 8, 10) in the array.

**11. Where is the bug?**

The function `double_it` is defined (not just declared) in the header file. If two `.cpp` files both `#include "helpers.h"`, the linker sees two definitions of `double_it` and reports a "multiple definition" error, violating the one-definition rule.

Fix it by adding the `inline` keyword:

```
inline int double_it(int n) {
    return n * 2;
}
```

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

The compiler produces a warning because `compute` is marked `[[nodiscard]]` and the return value of `compute(6, 7)` is discarded. The program still compiles, but the warning tells you that ignoring the result is almost certainly a bug.

**13. Pass `Album` by value vs by `const &`.**

Both versions compile and print the same thing, but `print_album_ref(const Album &a)` is preferred for a struct like `Album`.

The reason is *copying*. `print_album(Album a)` makes a complete copy of the `Album` struct every time it is called: a copy of the `int year`, plus full copies of the two `std::string` members (which each allocate and copy their character data). `print_album_ref(const Album &a)` does none of that — it just makes the parameter a reference to the existing `Album`, with no copy at all. The `const` is what makes it safe and self-documenting: the function promises not to modify the caller's `Album`, and the compiler enforces that promise.

For a function that takes a single `int` the answer is the *opposite*: just pass it by value. An `int` is the size of a CPU register; copying it is essentially free, and using a reference (`const int &`) actually adds a level of indirection that the function then has to follow. The rule of thumb is "pass cheap-to-copy types by value, pass everything else by `const &`." That cutoff varies by platform, but for this book the practical guidance is: built-in scalar types (`int`, `double`, `bool`, `char`, pointers) by value, and `std::string`, `std::vector`, structs, and other "owns memory" types by `const &`.

# Chapter 7: Numbers

**1. Convert the decimal number `200` to binary, hexadecimal, and octal by hand. Verify with a C++ program.**

- Binary: $200 = 128 + 64 + 8 = 2^7 + 2^6 + 2^3 =$ `11001000`

- Hex: 200 = 12 * 16 + 8 = `C8`
- Octal: 200 = 3 * 64 + 1 * 8 + 0 = `310`

```
#include <print>

int main() {
    int n = 200;
    std::println("Binary: {:b}", n);   // 11001000
    std::println("Hex:    {:x}", n);   // c8
    std::println("Octal:  {:o}", n);   // 310
}
```

**2. What does this print?**

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

`x` is binary `1100` = 12, and `y` is octal `52` = 5*8 + 2 = 42. The program prints `54`.

**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?**

With 8 bits there are $2^8 = 256$ distinct bit patterns to share between positive and negative values. Zero takes one of those slots and counts as a non-negative value, leaving 127 patterns for the strictly positive numbers (1 to 127) and 128 patterns for the negative numbers (-128 to -1). The asymmetry is the price you pay for having exactly one representation of zero.

**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;
}
```

`count` is unsigned, so it can never be negative. When `count` reaches `0` and you decrement, it underflows and wraps around to `UINT_MAX` (about 4.3 billion), which is still `>= 0`, so the loop continues forever. For an unsigned variable, the condition `count >= 0` is always true. Fix it by using a signed type (`int`), or by writing `while (count-- > 0)` to decrement after the test.

**5. Using two's complement with 8 bits, compute `100 - 75` by hand.**

```
 100 = 0110 0100
  75 = 0100 1011

Two's complement of 75:
       0100 1011
       1011 0100   (flip bits)
     + 0000 0001   (add 1)
       ---------
 -75 = 1011 0101

Add 100 + (-75):
       0110 0100   (100)
     + 1011 0101   (-75)
     -----------
     1 0001 1001
```

```
    ^
    overflow bit (discarded in 8 bits)
```

The result is `0001 1001` = 25, which is the correct answer for 100 - 75.

**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;
```

- `a` = `1 << 10` = $2^{10}$ = `1024`
- `b` = `100 >> 3` = 100 / 8 = `12` (integer division discards the remainder)
- `c` = `(1 << 4) - 1` = 16 - 1 = `15` (a common idiom for "n low bits all set")

**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);
```

The leading `0` makes `0700` an **octal** literal, not decimal. `0700` in octal equals 7 * 64 + 0 * 8 + 0 = `448` in decimal, so the program prints `Permissions: 448`. To get decimal 700, drop the leading zero: `int permissions = 700;`.

**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);
```

`big` fits in `int` (max is about 2.1 billion), but `big * 2` is 4 billion, which does **not** fit. This is **signed integer overflow** — undefined behavior. In practice many compilers will wrap to a negative value (you might see something like `-294967296`), but the standard does not require any particular result, and the compiler is free to do something else entirely. Use `long long` (or `int64_t`) for values that might exceed `int` range.

**9. Write a program that reads a hexadecimal color code (like `"FF8000"`), converts it to its red, green, and blue components, and prints each component in decimal and binary.**

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

int main() {
    std::print("Enter a hex color (e.g., FF8000): ");
    std::string color;
    std::cin >> color;

    int r = std::stoi(color.substr(0, 2), nullptr, 16);
    int g = std::stoi(color.substr(2, 2), nullptr, 16);
    int b = std::stoi(color.substr(4, 2), nullptr, 16);

    std::println("Red:   {:>3} ({:08b})", r, r);
    std::println("Green: {:>3} ({:08b})", g, g);
    std::println("Blue:  {:>3} ({:08b})", b, b);
}
```

**10. Without running it, determine the output of this program.**

```
uint8_t a = 250;
uint8_t b = 20;
```

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

Numerically, `a + b` is 270, but `sum` is `uint8_t` (8 bits, max 255), so the result wraps: 270 - 256 = `14`. There is also a display surprise: `std::println` with `{}` formats `uint8_t` (which is `unsigned char`) as a **character**, not a number. So instead of seeing `250 + 20 = 14`, you see the characters with codes 250, 20, and 14, which print as garbage or control characters on most terminals. To get numeric output, use `{:d}` or cast to `int`:

```
std::println("{} + {} = {}",
    static_cast<int>(a), static_cast<int>(b), static_cast<int>(sum));
// 250 + 20 = 14
```

**11. What does the digit-separator example print? Are the separators part of the value?**

It prints `1000000 16711935 61680`.

Digit separators (`'`) are *not* part of the stored value — they exist purely to make literals easier to read in the source code. The compiler ignores them entirely, so `1'000'000` is exactly the same value as `1000000`, `0xFF'00'FF` is exactly the same as `0xFF00FF`, and so on. You can place them anywhere between digits and group however you like.

**12. What does the `std::stoi` + `pos` chain print, and what is `pos` after each call?**

It prints `42 100 255`.

| Call | Returns | `pos` after the call |
|---|---|---|
| `std::stoi(input, &pos)` on `"42 100 255"` | 42 | 2 — index of the space after 42 |
| `std::stoi(input.substr(2), &pos)` on `" 100 255"` | 100 | 4 — 1 leading space + 100 is 4 chars |
| `std::stoi(input.substr(6))` on `" 255"` | 255 | (not requested) |

Each call skips leading whitespace, parses as much as it can, and stops at the first non-digit character. The `pos` parameter is how the caller learns where parsing stopped, which is what makes it possible to chain multiple parses through one string.

**13. Calculation: bitwise on 8-bit values.**

```
0b1010'1100 & 0b1111'0000 = 0b1010'0000 = 160
0b1010'1100 | 0b0000'1111 = 0b1010'1111 = 175
0b1010'1100 ^ 0b1111'1111 = 0b0101'0011 =  83
```

XOR with all-ones flips every bit — it is the same as the bitwise complement (`~`) for that width. The result is the *one's complement* of the original value.

**14. Calculation: `sizeof` and ranges on a typical 64-bit Linux system.**

| Type | `sizeof` | Largest unsigned value |
|---|---|---|
| `char` | 1 | 2^8 - 1 = 255 |
| `short` | 2 | 2^16 - 1 = 65,535 |
| `int` | 4 | 2^32 - 1 (~4.3 billion) |
| `long` | 8 | 2^64 - 1 (~1.8 * 10^19) |
| `long long` | 8 | 2^64 - 1 (~1.8 * 10^19) |

Note that on Linux/macOS `long` is 8 bytes but on 64-bit Windows `long` is still 4 bytes. This is exactly why `long` should be avoided when you need a specific width — use a fixed-width type like `int64_t` from `<cstdint>` instead.

**15. Write a program that prints an integer in decimal, hex, octal, binary, and as a string.**

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

int main()
{
    std::print("Enter an integer: ");
    int n{};
    std::cin >> n;

    std::println("decimal: {}",   n);
    std::println("hex:     {:#x}", n);
    std::println("octal:   {:#o}", n);
    std::println("binary:  {:#b}", n);

    std::string s = std::to_string(n);
    std::println("string:  \"{}\" (length {})", s, s.size());

    return 0;
}
```

The `#` flag adds the `0x`, `0`, and `0b` prefixes so the bases are obvious. `std::to_string` always produces the *decimal* string form of the number; if you want a hex string, use `std::format("{:x}", n)` instead.

# Chapter 8: Containers

**1. Think about it: Why does `std::array` require the size as part of its type while `std::vector` does not? What trade-off does this create?**

`std::array` stores its elements directly inside the object (on the stack), so the compiler needs to know the size at compile time to allocate the right amount of space. The size is part of the type, which means `std::array<int, 5>` and `std::array<int, 10>` are different types and cannot be assigned to each other.

`std::vector` stores its elements on the heap, and the size can change at runtime with `push_back` and `pop_back`. It does not need the size in its type.

The trade-off is that `std::array` is faster (no heap allocation) and has zero overhead, but it is inflexible — you must know the size at compile time. `std::vector` is more flexible but has a small overhead from heap allocation and potential reallocations.

**2. What does this print?**

```cpp
std::vector<int> v = {10, 20, 30};
v.push_back(40);
v.pop_back();
v.pop_back();
std::cout << v.size() << " " << v.back() << "\n";
```

It prints:

```
2 20
```

Starting with `{10, 20, 30}`, `push_back(40)` makes it `{10, 20, 30, 40}`. The first `pop_back()` removes 40: `{10, 20, 30}`. The second `pop_back()` removes 30: `{10, 20}`. The size is 2 and `back()` returns the last element, which is 20.

**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?**

3 more elements. The vector has room for 8 elements (capacity) and currently holds 5 (size), so it can accept 8 - 5 = 3 more elements before it needs to grow.

**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";
}
```

The loop condition uses `<=` instead of `<`. `scores.size()` is 3, so valid indices are 0, 1, and 2. When `i` is 3, `scores[3]` is an out-of-bounds access (undefined behavior). The fix is to use `<`:

```
for (int i = 0; i < scores.size(); i++) {
```

**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";
```

It prints:

```
5 10 15 20
```

The iterator loop visits each element from `begin()` to `end()`, printing each one.

**6. Think about it: Why is `for (auto x : vec)` (without `&`) generally a bad idea for vectors of strings? When would it be acceptable?**

Without `&`, each element is copied into `x` on every iteration. For `std::string`, this means allocating memory and copying the string data for each element, which is wasteful and slow.

It would be acceptable for small, cheap-to-copy types like `int`, `char`, or `double`, where copying is trivially fast. It could also be acceptable if you intentionally need a copy to modify independently of the original.

**7. Where is the bug?**

```
std::vector<std::string> playlist = {"Wannabe", "No Diggity"};
std::cout << playlist.at(2) << "\n";
```

The vector has 2 elements at indices 0 and 1. `playlist.at(2)` is out of bounds and will throw a `std::out_of_range` exception, crashing the program. The last valid index is 1.

**8. Calculation: A `std::vector<double>` contains 3 elements and has a capacity of 4. You call `push_back` 5 times. What is the size? What is the capacity?**

After 5 `push_back` calls, the size is 3 + 5 = **8**.

For capacity, starting at 4:

- Push 1 (size 4, capacity 4): fits
- Push 2 (size 5, capacity 4): exceeds capacity, doubles to 8
- Push 3 (size 6, capacity 8): fits
- Push 4 (size 7, capacity 8): fits
- Push 5 (size 8, capacity 8): fits

The final capacity is **8**.

**9. What does this print?**

```
std::vector<int> v = {1, 2, 3};
v.clear();
std::cout << v.size() << " " << v.empty() << "\n";
```

It prints:

```
0 1
```

`v.clear()` removes all elements, making the size 0. `v.empty()` returns `true`, which prints as `1`.

**10. Write a program that reads numbers from the user (enter -1 to stop), stores them in a `std::vector<int>`, and prints them in reverse order.**

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

int main()
{
    std::vector<int> numbers;
    int n;

    std::cout << "Enter numbers (-1 to stop):" << std::endl;

    while (std::cin >> n && n != -1) {
        numbers.push_back(n);
    }

    std::cout << "In reverse:" << std::endl;
    for (int i = static_cast<int>(numbers.size()) - 1; i >= 0; i--) {
        std::cout << numbers[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

**11. What does this print?**

The vector starts as `{10, 20, 30, 40, 50}`. After `insert(v.begin() + 2, 25)`, it becomes `{10, 20, 25, 30, 40, 50}`. After `erase(v.begin())`, it becomes `{20, 25, 30, 40, 50}`.

It prints: `20 25 30 40 50`

**12. Calculation:** What is the size and capacity after calling `reserve(100)` on an empty `std::vector<int>`, then calling `push_back` 3 times?

The size is 3 (three elements were added). The capacity is at least 100 (the `reserve` call preallocated room for 100 elements, and adding 3 elements does not exceed that, so no reallocation occurs).

**13. Where is the bug? `push_back` inside an iterator loop.**

The bug is **iterator invalidation**. When `push_back` runs out of capacity, the vector reallocates its storage to a new (larger) buffer and copies the elements over. After that happens, every iterator, pointer, and reference into the old storage — including the loop variable `it` — is dangling. The next `++it` and `*it` then access freed memory, which is undefined behavior.

You may "get away with it" sometimes if the vector happens to have spare capacity at the moment you call `push_back`, but the moment it has to grow, the loop blows up.

Two safe ways to fix it:

```
// 1. Collect the work first, then apply it after the loop ends.
std::vector<int> to_add;
for (int x : v) {
    if (x == 3) to_add.push_back(99);
}
for (int x : to_add) v.push_back(x);

// 2. Use indices, and re-read v.size() each iteration. Indices are
//    not invalidated by reallocation the way iterators are.
for (std::size_t i = 0; i < v.size(); ++i) {
    if (v[i] == 3) v.push_back(99);
}
```

The general rule: do not modify a container's structure while iterating over it. If you need to add or remove elements based on what you find, build a list of changes first and apply them afterward.

**14. Think about it: `std::array<double, 3>` vs `std::vector<double>`.**

The size `3` is part of the *type* of `std::array`. A function that takes `const std::array<double, 3> &` will only accept a 3-element array; passing a `std::array<double, 4>` is a compile error, because `std::array<double, 3>` and `std::array<double, 4>` are completely different types. That is exactly the point: the compiler statically guarantees that the function will receive 3 doubles, no more, no less.

`const std::vector<double> &` is more flexible — it accepts any size — but the cost is that the function has to check `v.size()` at run time and decide what to do if it is not 3. There is no compile-time guarantee that the input is the right shape.

Use `std::array<T, N>` when the size is known and fixed at compile time and you want the compiler to enforce it. Use `std::vector<T>` when the size is determined at run time or might change.

**15. Calculation: `reserve(8)` then 12 `push_back` calls.**

After `reserve(8)`: `size() == 0`, `capacity() == 8`.

| `push_back` # | size | capacity | reallocated? |
|---|---|---|---|
| 1 | 1 | 8 | no |
| 2 | 2 | 8 | no |
| 3 | 3 | 8 | no |
| 4 | 4 | 8 | no |
| 5 | 5 | 8 | no |
| 6 | 6 | 8 | no |
| 7 | 7 | 8 | no |
| 8 | 8 | 8 | no |
| 9 | 9 | 16 | **yes** |
| 10 | 10 | 16 | no |
| 11 | 11 | 16 | no |
| 12 | 12 | 16 | no |

The 9th `push_back` is the only one that reallocates: capacity was full at 8, and the doubling rule grows it to 16. That is also the only call where existing iterators, pointers, and references into the vector are invalidated. The 1st through 8th calls only write into already-reserved storage, and the 10th through 12th only fill in the freshly-allocated space, so iterators obtained *after* the reallocation are still valid.

# Chapter 9: I/O Streams

**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;
}
```

It prints:

```
10 + 20 = 30
```

The `ostringstream` builds the string piece by piece. The expression `10 + 20` is evaluated to 30 before being streamed.

**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;
}
```

It prints:

```
3.14 100 hola
```

The `>>` operator extracts values in order from the string: `n` gets 100, `s` gets "hola", `d` gets 3.14. The `cout` statement prints them in a different order: `d`, `n`, `s`.

**3. What is wrong with this code?**

```cpp
#include <fstream>
#include <iostream>

int main()
{
    std::ifstream infile("data.txt");
    std::string line;

    while (std::getline(infile, line)) {
        std::cout << line << std::endl;
```

```
    }

    return 0;
}
```

The code does not check whether the file opened successfully before reading from it. If `data.txt` does not exist, `infile` will be in a failed state, `std::getline` will immediately return false, and the program will silently produce no output with no error message. The fix is to check the stream after opening:

```
std::ifstream infile("data.txt");
if (!infile) {
    std::cerr << "Could not open data.txt" << std::endl;
    return 1;
}
```

**4. What is wrong with this file-writing code?**

```
#include <fstream>

int main()
{
    std::ofstream out;
    out << "Yo me la paso bien" << std::endl;
    out.close();
    return 0;
}
```

The `std::ofstream` is declared but never given a filename. The stream is not connected to any file, so writing to it does nothing. The fix is to pass a filename to the constructor or call `.open()`:

```
std::ofstream out("output.txt");
```

**5. Why is it useful that string streams, file streams, and `std::cout`/`std::cin` all share the same `<<` and `>>` interface?**

Because they share the same interface, code written to work with one type of stream can work with any type of stream. For example, a function that writes data using `<<` on an `std::ostream&` reference can write to the screen (`std::cout`), to a file (`std::ofstream`), or to a string (`std::ostringstream`) without any changes. This makes code more flexible and reusable.

**6. Write a program that reads three song names from the user, builds a single string with them separated by `/` using an `std::ostringstream`, writes it to `favorites.txt`, then reads the file back and prints its contents.**

```
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    std::ostringstream oss;

    for (int i = 0; i < 3; i++) {
        std::string song;
        std::cout << "Song name: ";
        std::getline(std::cin, song);
        if (i > 0) {
            oss << " / ";
```

```
        }
        oss << song;
    }

    std::ofstream outfile("favorites.txt");
    if (!outfile) {
        std::cerr << "Could not open favorites.txt" << std::endl;
        return 1;
    }
    outfile << oss.str() << std::endl;
    outfile.close();

    std::ifstream infile("favorites.txt");
    if (!infile) {
        std::cerr << "Could not read favorites.txt" << std::endl;
        return 1;
    }

    std::string line;
    while (std::getline(infile, line)) {
        std::cout << line << std::endl;
    }
    infile.close();

    return 0;
}
```

**7. What does this program print?**

It prints:

```
true
3.1
```

`std::boolalpha` makes the `bool` value `true` print as `true` instead of `1`. `std::fixed` combined with `std::setprecision(1)` formats the `double` with exactly 1 digit after the decimal point.

**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?**

With `std::ios::app`, new data is appended to the end of the file. The existing content is preserved.

By default, `std::ofstream` opens with `std::ios::out | std::ios::trunc`, which truncates (erases) the file before writing. Any existing content is lost.

**9. Given the input string `"Closing Time 1998 Smooth 1999"`, how many times will this loop iterate? What is the final value of `count`?**

The loop iterates **5** times, and the final value of `count` is **5**. `iss >> word` reads one whitespace-delimited token per iteration: “Closing”, “Time”, “1998”, “Smooth”, “1999”. The `>>` operator splits on whitespace, so each word and number is a separate token.

**10. What does the bidirectional `std::stringstream` example print?**

It prints `[year] [1999]`.

After `ss << "year " << 1999;`, the stream contains the characters `year 1999` in its internal buffer. The first `ss >> word` reads up to the next whitespace, so `word` becomes `"year"` and the stream’s read position now sits on the space. The second `ss >> year` skips the leading whitespace and parses `1999` into the `int`

`year`. A `std::stringstream` is just a `std::ios::in | std::ios::out` stream backed by a string, so the same object can both produce text (via `<<`) and consume text (via `>>`).

**11. Calculation: file open mode flags.**

Each flag is one bit in a `std::ios_base::openmode` bitmask, so combinations are made by OR'ing them together. You use `|` because that is the bitwise OR operator (the same one introduced in Chapter 7) — it sets all the bits that are on in either operand, which is exactly what "this mode AND that mode" means. You cannot use `+` or `,` because those would either give the wrong numeric value or not produce a valid `openmode` at all.

| Expression | What it asks for |
|---|---|
| `std::ios::out` | open for writing (the default for `ofstream`) |
| `std::ios::out \| std::ios::app` | open for writing in **append** mode (do not truncate) |
| `std::ios::out \| std::ios::trunc` | open for writing and **truncate** the file to zero length |
| `std::ios::in \| std::ios::out \| std::ios::binary` | open for reading **and** writing in **binary** mode (no newline translation) |

**12. Write a program that reads `oldies.txt`, prints each line, and reports errors on `std::cerr`.**

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

int main()
{
    std::ifstream in("oldies.txt");
    if (!in) {
        std::cerr << "could not open oldies.txt for reading\n";
        return 1;
    }

    std::vector<std::string> lines;
    std::string line;
    while (std::getline(in, line)) {
        lines.push_back(line);
    }

    for (const std::string &l : lines) {
        std::cout << l << "\n";
    }
    return 0;
}
```

`std::cerr` matters even though it goes to the same terminal by default for two reasons. First, `std::cerr` is **unbuffered**, so error messages reach the user immediately even if the program is about to crash. Second, on the command line `std::cout` and `std::cerr` are independently redirectable: a user running `./program > out.txt` only sends standard output to the file, so error messages still appear on the screen. If the program had written errors to `std::cout` instead, they would have ended up silently in `out.txt`. Sending data to `std::cout` and errors to `std::cerr` is the convention every Unix tool follows for exactly this reason.

# Chapter 10: std::format and std::print

**1. What does `std::format("{:>8.2f}", 3.1)` produce? How many characters wide is the result?**

It produces "    3.10". The format specifier `>8.2f` means: right-align in a field 8 characters wide, with 2 decimal places. `3.1` formatted with 2 decimal places becomes `3.10`, which is 4 characters. Right-aligned in 8 characters, it is padded with 4 spaces on the left. The result is **8 characters wide**.

**2. Why might you prefer `std::format` over chaining `<<` operators with `std::cout`?**

1. **Readability:** A format string like `std::format("{} scored {} points", name, score)` is much easier to read than `std::cout << name << " scored " << score << " points"`. The format string shows the complete output pattern in one place.
2. **Formatting control:** Width, alignment, and precision are specified concisely inside `{}` placeholders (e.g., `{:>10.2f}`), rather than using verbose manipulators like `std::setw`, `std::setprecision`, and `std::setfill`.

**3. What is the difference between `std::print` and `std::println`?**

`std::print` prints formatted output without a trailing newline. `std::println` prints formatted output followed by a newline. They both use the same format string syntax as `std::format`.

**4. What does `std::format("{:*^20}", "Hola")` produce?**

It produces `"********Hola********"`. The format specifier `*^20` means: center-align in a field 20 characters wide, filling with `*` characters. "Hola" is 4 characters, so it gets 8 `*` characters on each side.

**5. What is wrong with this code?**

```
std::string result = std::format("{} scored {1} points", name, score);
```

You cannot mix implicit (`{}`) and indexed (`{1}`) argument references in the same format string. The fix is to use either all implicit or all indexed:

```
std::string result = std::format("{} scored {} points", name, score);
// or
std::string result = std::format("{0} scored {1} points", name, score);
```

**6. Write a program that asks for three song names and scores, writes them to a file, reads the file back, and prints a formatted table.**

```
#include <fstream>
#include <format>
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    std::ofstream outfile("rankings.txt");
    if (!outfile) {
        std::cerr << "Could not open rankings.txt for writing" << std::endl;
        return 1;
    }

    for (int i = 0; i < 3; i++) {
        std::string song;
        double score;

        std::cout << "Song name: ";
        std::getline(std::cin, song);
        std::cout << "Score: ";
        std::cin >> score;
```

```
        std::cin.ignore();

        outfile << song << "|" << score << std::endl;
    }
    outfile.close();

    std::ifstream infile("rankings.txt");
    if (!infile) {
        std::cerr << "Could not open rankings.txt for reading" << std::endl;
        return 1;
    }

    std::cout << std::format("{:<25} {:>6}", "Song", "Score") << std::endl;
    std::cout << std::string(32, '-') << std::endl;

    std::string line;
    while (std::getline(infile, line)) {
        size_t sep = line.find('|');
        std::string song = line.substr(0, sep);
        double score = std::stod(line.substr(sep + 1));
        std::cout << std::format("{:<25} {:>6.1f}", song, score) << std::endl;
    }

    infile.close();
    return 0;
}
```

**7. What does the indexed-placeholder example print?**

The first call prints:

```
I Want It That Way - Backstreet Boys (1999)
```

The placeholders `{1} - {0} ({2})` reorder the arguments at format time: `{0}` is the first argument (`"Backstreet Boys"`), `{1}` is the second (`"I Want It That Way"`), `{2}` is the third (`1999`).

Replacing every placeholder with `{}` and using the original argument order prints:

```
Backstreet Boys - I Want It That Way (1999)
```

The rule is: in any single format string, all placeholders must be either *all implicit* (`{}`, `{}`, `{}`) or *all indexed* (`{0}`, `{1}`, `{2}`). You cannot mix the two styles. A format string like `"{} - {1} - {2}"` is rejected at compile time — pick one style and stick with it.

**8. Calculation: sign and zero-padding format specs.**

| Expression | Result | Notes |
|---|---|---|
| `std::format("{:+d}", 42)` | `"+42"` | `+` flag forces a sign on positive numbers |
| `std::format("{:+d}", -42)` | `"-42"` | negatives still get a `-` |
| `std::format("{: d}", 42)` | `" 42"` | space flag puts a space where `+` would go on positives |
| `std::format("{:05d}", 42)` | `"00042"` | width 5, zero-padded; `42` becomes `00042` |
| `std::format("{:+06d}", -42)` | `"-0042"` | width 6, zero-padded, signed; the `-` counts toward the width |

The width includes the sign character, so `"{:+06d}"` on `-42` is `-` followed by 4 zeros and then `42`, totaling 6 characters.

**9. What does the `#` and `08b` example print, and what do those flags mean?**

It prints:

```
0xff
0377
0b11111111
11111111
```

- `#` is the **alternate form** flag. For `x` it prepends `0x`, for `o` it prepends `0`, and for `b` it prepends `0b`. Without `#` you would just see `ff`, `377`, and `11111111`.
- `08b` means width 8, zero-padded, base 2. `255` already takes 8 bits to write in binary, so the result is `11111111` with no padding needed. If the value were `5`, the same spec would produce `00000101`.

**10. What does the string-precision example print, and what does precision mean for strings?**

It prints:

```
[Smell]
[Smell     ]
[     Smell]
```

For a string argument, precision (`.5`) is the **maximum number of characters to use from the string** — it truncates anything longer. That is the opposite of how precision works for floating-point numbers, where `.5` means "show 5 digits after the decimal point" (which can make the result *longer*, not shorter).

Combined with width and alignment:

- `{:.5}` is just "at most 5 characters", with no width.
- `{:<10.5}` is "at most 5 characters, left-aligned in a field of width 10", padded with spaces on the right.
- `{:>10.5}` is the same thing right-aligned.

So the string is first truncated to `"Smell"` and then placed in a 10-wide field.

**11. Write a program that prints three integers in decimal, hex, and binary.**

```cpp
#include <print>
#include <format>
#include <string>

void show(int n)
{
    std::string row = std::format(
        "{:>6d}  {:#010x}  {:#018b}",
        n, n, n);
    std::println("{}", row);
}

int main()
{
    show(42);
    show(255);
    show(65'535);
    return 0;
}
```

The format specs do all the work:

- `{:>6d}` — decimal, right-aligned in a 6-character field.
- `{:#010x}` — hex with the `0x` prefix (`#`), zero-padded (`0`) to a total field width of 10. Eight of those characters are the hex digits, the other two are `0x`.
- `{:#018b}` — binary with the `0b` prefix, zero-padded to a total field width of 18 (16 bits + the 2-character `0b`).

Output:

```
    42  0x0000002a  0b0000000000101010
   255  0x000000ff  0b0000000011111111
 65535  0x0000ffff  0b1111111111111111
```

Notice how a single `std::format` call combines width, alignment, fill, the alternate-form flag, zero padding, and the base specifier all in the same string.

# Chapter 11: Exceptions

### 1. What does the following program print?

```
#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;
}
```

It prints:

```
B: oops
C
```

`step1()` calls `step2()`, which calls `step3()`, which throws. The exception propagates back through `step2` and `step1` to the `catch` block in `main`. `"A"` is never printed because the rest of the `try` block is skipped. After the `catch` block handles the exception, execution continues normally and `"C"` is printed.

### 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;
}
```

The `catch (const std::invalid_argument &e)` block will never execute. `std::invalid_argument` derives from `std::exception`, and `catch` blocks are tried in order. The `catch (const std::exception &e)` block matches any `std::exception` (including `std::invalid_argument`), so it catches the exception before the more specific handler gets a chance. The fix is to put more specific `catch` blocks before more general ones — move `std::invalid_argument` above `std::exception`.

**3. Why should you always catch exceptions by `const` reference rather than by value?**

Catching by value makes a copy of the exception object, which can **slice** it. If the thrown exception is a derived type (like `std::out_of_range`) and you catch by value as `std::exception`, the copy loses the derived class's data — you get only the base class portion. Catching by `const` reference avoids the copy and preserves the full object, including any derived-class behavior. The `const` part signals that you do not intend to modify the exception.

**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;
}
```

It prints:

```
Marshall on
Fender on
Fender off
Marshall off
handled
```

The two `Amp` objects are constructed in order. When the exception is thrown, the stack unwinds and destroys them in reverse order — `Fender` first, then `Marshall`. After stack unwinding, the `catch (...)` block runs.

**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");
}
```

Yes, it compiles. The compiler does not check whether a `noexcept` function actually avoids throwing — `noexcept` is a promise, not a compile-time guarantee. When `play()` is called, `load()` throws `std::runtime_error`. Because `play()` is marked `noexcept`, the exception cannot escape it, so the program calls `std::terminate()` and crashes immediately — no chance to catch the exception.

**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;
}
```

It prints:

```
3
division by zero
```

`divide(10, 3)` returns the expected value `3` (integer division truncates). `divide(10, 0)` returns an unexpected error `"division by zero"`. The boolean check `if (r1)` is true because `r1` holds a value; `if (!r2)` is true because `r2` holds an error.

**7. When would you use `std::expected` instead of throwing an exception? Give an example scenario for each.**

Use `std::expected` when failure is a *normal, expected outcome* that the caller will handle immediately. For example, parsing user input: if you ask the user for a number and they type "abc", that is not exceptional — it is a routine case. Returning `std::expected<int, std::string>` lets the caller inspect the error and try again.

Use exceptions when failure is *rare and should propagate* up several layers. For example, opening a configuration file that the program requires: if the file is missing, the error should propagate up to a high-level handler that can report the problem and shut down gracefully. Threading error codes through every intermediate function would be tedious and error-prone.

**8. How many destructors run before the `catch` block executes?**

Three destructors run before the `catch` block. When `inner()` throws, stack unwinding destroys `b` ("Vogue") and then `a` ("Torn") in reverse order of construction. Then `outer()`'s frame unwinds, destroying `c` ("Iris"). Only after all three destructors complete does the `catch` block execute and print "caught".

**9. Write a function `safe_sqrt` that returns `std::expected<double, std::string>`.**

```
#include <cmath>
#include <expected>
#include <iostream>
#include <string>

std::expected<double, std::string> safe_sqrt(double x) {
    if (x < 0) {
        return std::unexpected("cannot take square root of negative number");
    }
    return std::sqrt(x);
}

int main()
{
    auto r1 = safe_sqrt(25.0);
    if (r1) {
        std::cout << "sqrt(25) = " << *r1 << std::endl;
    }

    auto r2 = safe_sqrt(-4.0);
    if (!r2) {
        std::cout << "Error: " << r2.error() << std::endl;
    }

    return 0;
}
```

Output:

```
sqrt(25) = 5
Error: cannot take square root of negative number
```

**10. Where is the bug? `catch(...)` before `catch(const std::out_of_range &)`.**

The program prints `anything`, and the `out_of_range` handler is never reached.

`catch` clauses are matched **in source order**, top to bottom. `catch(...)` matches *anything*, so once it appears at the top of the list, no later handler can ever fire — the more specific `catch(const std::out_of_range &)` is dead code.

The fix is to put the most specific handlers first and `catch(...)` last as a final safety net:

```
try {
    throw std::out_of_range("nope");
}
catch (const std::out_of_range &e) {
    std::cout << "out_of_range: " << e.what() << "\n";
}
catch (const std::exception &e) {       // any other std exception
    std::cout << "std::exception: " << e.what() << "\n";
}
```

```
catch (...) {                               // truly unknown
    std::cout << "unknown exception\n";
}
```

This is the standard layering: type-specific, then `std::exception` for everything from the standard library, then `catch(...)` to make sure no exception escapes the function. With this ordering, the program now prints `out_of_range: nope`.

**11. Write `parse_age` that translates `std::stoi` failures into specific exceptions.**

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

int parse_age(const std::string &s)
{
    int n = 0;
    try {
        n = std::stoi(s);
    }
    catch (const std::invalid_argument &) {
        throw std::invalid_argument("not a number");
    }
    catch (const std::out_of_range &) {
        throw std::out_of_range("age must be 0..150");
    }

    if (n < 0 || n > 150) {
        throw std::out_of_range("age must be 0..150");
    }
    return n;
}

int main()
{
    for (const std::string &s : {"42", "abc", "-1"}) {
        try {
            int age = parse_age(s);
            std::cout << s << " -> " << age << "\n";
        }
        catch (const std::invalid_argument &e) {
            std::cout << s << " -> invalid: " << e.what() << "\n";
        }
        catch (const std::out_of_range &e) {
            std::cout << s << " -> range: " << e.what() << "\n";
        }
    }
    return 0;
}
```

Output:

```
42 -> 42
abc -> invalid: not a number
-1 -> range: age must be 0..150
```

The `parse_age` function catches `std::stoi`'s exceptions and re-throws them with our own messages, then

does the `[0, 150]` range check itself. The caller in `main` distinguishes the two error categories with separate `catch` clauses, so different error types get different messages without using a single generic `catch(...)`.

# Chapter 12: Classes

**1. What is the difference between a `struct` and a `class` in C++? Why would you choose one over the other?**

The only technical difference is the default access level. Members of a `struct` are `public` by default, while members of a `class` are `private` by default.

By convention, `struct` is used for simple data holders with public members (plain old data). `class` is used when you want to encapsulate data with behavior — bundling private data with public member functions that control access.

**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;
}
```

It prints:

```
Metallica arrives
Soundgarden arrives
show time
Soundgarden exits
Metallica exits
```

Objects are constructed in order of declaration. Destructors are called in reverse order when the objects go out of scope at the end of `main()`. So `b` (Soundgarden) is destroyed before `a` (Metallica).

**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; }
};
```

The `increment()` function is marked `const`, but it modifies the member variable `count`. A `const` member function promises not to modify the object, so `count++` inside a `const` function is a compilation error. The fix is to remove `const` from `increment()`:

```
void increment() {
    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.**

Member initializer lists initialize members directly, while assignment in the constructor body first default-constructs the members and then assigns new values. For complex types like `std::string`, the initializer list avoids the wasted work of constructing a default value that is immediately overwritten.

An initializer list is *required* for `const` members and reference members because they cannot be assigned to after construction:

```
class Example {
    const int id;
    int &ref;
public:
    // Must use initializer list - cannot assign to const or ref in body
    Example(int i, int &r) : id(i), ref(r) {}
};
```

**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?**

Three `int` members at 4 bytes each = 12 bytes. One `std::string` at 32 bytes. Total minimum: 12 + 32 = **44 bytes** (ignoring padding).

**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;
}
```

It prints:

```
1
0
```

`a == b` compares titles: `"All Star" == "All Star"` is true, which prints as 1. `a == c` compares titles: `"All Star" == "Enter Sandman"` is false, which prints as 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;
    }
};
```

The constructor parameters have the same names as the member variables, so the parameters shadow the members. The line `name = name` will not compile because the parameter `name` is `const std::string &` — you cannot assign to a const reference. Even if both parameters were non-const, the assignments would just assign each parameter to itself without ever setting the members.

The fix is to use `this->` or, better yet, a member initializer list:

```
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;
}
```

It prints:

```
Playing: Torn
Tuned to station 98
Playing: Basket Case at volume 11
```

The compiler matches each call to the overload whose parameters match the arguments. `r.play("Torn")` matches the `string` overload, `r.play(98)` matches the `int` overload, and `r.play("Basket Case", 11)` matches the `string, int` overload.

**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;
};
```

The call `set_volume(50)` is ambiguous. It could match either `set_volume(int)` or `set_volume(int, int)` (using the default value of 100 for `max`). The compiler cannot decide which one to call and will refuse to compile the code. The fix is to remove one of the overloads or change the default parameter design so the signatures do not overlap.

**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?**

Default parameters must appear at the end because the compiler fills in defaults from right to left. If a non-default parameter came after a default one, there would be no way to skip the default and supply the later argument.

For example, `void f(int a = 10, int b)` would make `f(5)` ambiguous — is 5 the value for `a` or `b`? The compiler rejects this as an error.

**11. What is wrong with this code?**

The `TrackNumber` constructor takes a single `int` and is not marked `explicit`. This means the compiler silently converts `7` to `TrackNumber(7)` when calling `play(7)`. The code compiles and runs, but the implicit conversion is surprising — the caller probably meant to pass an integer, not construct a `TrackNumber`.

Fix it by adding `explicit`:

```
explicit TrackNumber(int n) : number(n) {}
```

Now `play(7)` will not compile, and the caller must write `play(TrackNumber(7))`.

**12. What does `explicit operator bool()` allow that a non-explicit `operator bool()` also allows? What does it prevent?**

Both versions allow the object to be used in boolean contexts like `if (obj)`, `while (obj)`, and `!obj`. These are called "contextual conversions to `bool`" and work even with `explicit`.

The `explicit` version prevents the object from being used in arithmetic, comparisons with integers, or other contexts that would silently convert it to `bool` (and then to `int`). For example, without `explicit`, `obj + 5` would compile — `obj` converts to `bool` (`true` = 1), and then `1 + 5` gives 6. With `explicit`, that expression is a compile error.

**13. Write a class called `Counter`.**

```
#include <iostream>

class Counter {
private:
    int count;

public:
    Counter() : count(0) {}

    void increment() { count++; }
    void reset() { count = 0; }
    int value() const { return count; }

    bool operator==(const Counter &other) const {
        return count == other.count;
    }
};

int main()
{
    Counter a, b;
    a.increment();
    a.increment();
    a.increment();
    std::cout << "a: " << a.value() << std::endl;  // 3

    b.increment();
    b.increment();
    b.increment();
    std::cout << "a == b: " << (a == b) << std::endl;  // 1 (true)

    a.reset();
    std::cout << "a after reset: " << a.value() << std::endl;  // 0
    std::cout << "a == b: " << (a == b) << std::endl;  // 0 (false)

    return 0;
}
```

**14. Where is the bug? `p.tracks.push_back(...)`.**

The compiler rejects the program with something like:

```
error: 'std::vector<...> Playlist::tracks' is private within this context
```

`tracks` is in the `private:` section of `Playlist` (the default for `class`), so code outside of `Playlist` cannot touch it. This is exactly the design rule access specifiers exist to enforce: the only way to interact with a class's data is through its `public` members.

The smallest change that compiles is to make `tracks` public:

```
class Playlist {
public:
    std::vector<std::string> tracks;
    int size() const { return tracks.size(); }
};
```

That works but throws away encapsulation — `Playlist` no longer controls how its tracks are added or removed. The *better* fix is to keep `tracks` private and add a real public method that mediates access:

```
class Playlist {
    std::vector<std::string> tracks;
public:
    void add(const std::string &track) { tracks.push_back(track); }
    int  size() const { return tracks.size(); }
};
```

Now `Playlist` keeps full control over its internal vector and can later add validation, logging, or change the underlying storage without breaking any caller.

**15. Write a `Builder` with a chainable `add` method.**

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

class Builder {
    std::vector<int> values;
public:
    Builder &add(int v) {
        values.push_back(v);
        return *this;
    }

    void print() const {
        for (int v : values) {
            std::cout << v << " ";
        }
        std::cout << "\n";
    }
};

int main()
{
    Builder b;
    b.add(1).add(2).add(3).add(4);
    b.print();   // 1 2 3 4
    return 0;
}
```

`add` returns `*this` **by reference** so each call hands the *same* `Builder` back to the next call in the chain. If `add` returned `*this` by value, each call would build a fresh copy and the chain would be acting on temporary objects — the original `b` would never get any of the values added past the first one. Returning a reference is what makes the fluent interface actually fluent.

**16. Think about it: `explicit operator bool()`.**

`operator bool` is marked `explicit` so the conversion only happens in places where the compiler is *expecting* a bool, not anywhere a `Volume` happens to be used in arithmetic or assignment. That avoids the classic "safe bool" footgun where a bool conversion accidentally enables nonsensical comparisons like `volume + 1` or `volume == otherVolume`.

| Call | Compiles? | Why |
|---|---|---|
| `(a) if (v) { ... }` | yes | The condition of `if` is a "contextual" bool conversion, which `explicit` allows. |
| `(b) bool b = v;` | **no** | Implicit copy-initialization to `bool` is not contextual, so `explicit` blocks it. |
| `(c) bool b2 = static_cast<bool>(v);` | yes | An explicit cast asks for the conversion by name, which is exactly what `explicit` permits. |
| `(d) int n = v;` | **no** | There is no `operator int`, only `operator bool`, and `bool` does not implicitly convert to `int` here without first going through the explicit cast. |

So `explicit operator bool()` lets you write `if (v)`, `while (v)`, `!v`, and so on, while preventing the conversion from sneaking in where you didn't ask for it.

# Chapter 13: Memory Management

**1. What is the difference between stack and heap memory? Give one situation where you would need to use the heap.**

Stack memory is automatically managed — variables are created when declared and destroyed when they go out of scope. Stack allocation is fast but limited in size and lifetime.

Heap memory is manually managed (or managed through smart pointers). It persists until explicitly freed and can be much larger than the stack.

You would need the heap when you need memory to outlive the current scope (e.g., creating an object inside a function and returning a pointer to it), or when the size of the data is not known at compile time (e.g., reading an unknown number of records from a file).

**2. What does the following program print?**

```cpp
#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;
}
```

It prints:

```
3
2
1
```

After creating `p`, `q`, and `r` all pointing to the same object, the reference count is 3. `q.reset()` releases q's ownership, dropping the count to 2. `r.reset()` releases r's ownership, dropping the count to 1. Only `p` still owns the object.

**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;
}
```

The array was allocated with `new int[3]` (array `new`), but freed with `delete` (non-array `delete`). When you allocate with `new[]`, you must free with `delete[]`. Using plain `delete` on an array is undefined behavior. The fix:

```
delete[] volumes;
```

**4. Why can you not copy a `std::unique_ptr`? What should you do instead if you want to transfer ownership?**

A `std::unique_ptr` represents sole ownership of a resource. If you could copy it, two `unique_ptr`s would own the same memory, and both would try to delete it when destroyed, causing a double-free bug.

To transfer ownership, use `std::move`:

```
std::unique_ptr<int> a = std::make_unique<int>(42);
std::unique_ptr<int> b = std::move(a);  // ownership transferred to b
// a is now nullptr
```

**5. After `std::move(a)` is called, is it safe to use `a`? What state is `a` in?**

After `std::move(a)`, `a` is in a valid but unspecified state. It is safe to assign a new value to `a` or to destroy it, but you should not read its value or call methods that depend on its contents. For `std::string`, the moved-from string is typically empty. For `std::unique_ptr`, the moved-from pointer is `nullptr`.

**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;
}
```

Both `a` and `b` are constructed from the same raw pointer, so they both think they own the same memory. When they go out of scope, both will try to `delete` the same pointer, resulting in a double-free bug (undefined behavior). This is why you should use `std::make_unique` instead of constructing `unique_ptr` from raw pointers, and never give the same raw pointer to two smart pointers.

**7. If a `std::shared_ptr` is copied 4 times (so there are 5 `shared_ptrs` total), what is the reference count? How many need to be destroyed before the object is freed?**

The reference count is **5**. All 5 `shared_ptrs` must be destroyed (or reset) before the object is freed. The object is deleted when the last `shared_ptr` owning it is destroyed, which brings the reference count from 1 to 0.

**8. Write a program with `std::unique_ptr` that demonstrates moving ownership.**

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

int main()
{
    std::unique_ptr<std::string> first = std::make_unique<std::string>("Wannabe");
    std::cout << "first: " << *first << std::endl;

    std::unique_ptr<std::string> second = std::move(first);
    std::cout << "second: " << *second << std::endl;

    if (!first) {
        std::cout << "first is empty (nullptr)" << std::endl;
    }

    return 0;
}
```

Output:

```
first: Wannabe
second: Wannabe
first is empty (nullptr)
```

**9. What does the following code print?**

```
int x = 10;
int *p = &x;
*p = 20;
std::cout << x << std::endl;
```

It prints `20`. `p` points to `x`, so `*p = 20` modifies `x` through the pointer.

**10. Given a `struct Song` and a pointer `Song *ptr`, write two equivalent expressions to access `title`.**

```
(*ptr).title   // dereference first, then access member
ptr->title     // arrow operator --- same thing, cleaner
```

Both expressions access the `title` member of the `Song` that `ptr` points to. `ptr->title` is the preferred form.

**11. Where is the bug? `play(Song *song)` with a `nullptr`.**

`play` dereferences `song` (`song->title`) without checking that the pointer is non-null first. Calling it with `nullptr` reads from address 0, which is undefined behavior — on most systems it will crash with a segmentation fault.

The smallest safe change is to check the pointer at the top of the function and either return early or throw, so the program never dereferences a null pointer:

```
void play(Song *song)
{
    if (song == nullptr) {
        std::cout << "(no song)\n";
        return;
    }
    std::cout << song->title << " (" << song->year << ")\n";
}
```

The deeper fix is to use a reference (`Song &`) instead of a pointer when the function never wants to handle "no song", since references cannot be null and the caller is forced to provide a real `Song`.

**12. Think about it: What is RAII?**

RAII — *Resource Acquisition Is Initialization* — is the C++ idiom that ties the lifetime of a resource to the lifetime of an object on the stack. The constructor of the object **acquires** the resource (memory, a file handle, a lock, etc.), and the destructor **releases** it. Because C++ guarantees that destructors run when an object goes out of scope — whether that happens normally, via an early `return`, or because an exception was thrown — you cannot accidentally forget to release the resource.

`std::unique_ptr<T>` is the RAII wrapper around `new T(...)` / `delete` for heap memory. Its constructor takes ownership of a freshly `new`-allocated pointer, and its destructor calls `delete` for you. You never write the matching `delete` yourself, and you cannot leak the memory by taking an early `return`.

Two other RAII types you have already seen:

- **`std::vector<T>`** — the vector's destructor frees the heap buffer that holds its elements.
- **`std::ofstream` / `std::ifstream`** — the destructor closes the underlying file handle, so you do not have to call `.close()` yourself in normal code paths.

Both follow the same pattern: a stack object that owns something on the heap or in the OS, and tears it down automatically when its scope ends.

**13. Where is the bug? `make_playlist` leak.**

The early-return path leaks:

```
if (fav->size() > 100) {
    return;             // <-- fav is never deleted
}
```

If the string is longer than 100 characters, the function returns *without* running `delete fav`, and the heap object lives forever (until the process exits). Adding another `delete` before the `return` would fix this one path, but the next time someone adds a new return statement they would have to remember to do the same thing.

The correct fix is to stop using raw `new` for owning the heap object and let RAII do the cleanup:

```
#include <memory>

void make_playlist()
```

```
{
    auto fav = std::make_unique<std::string>("Wonderwall");
    if (fav->size() > 100) {
        return;        // unique_ptr's destructor frees the string here
    }
    std::cout << *fav << "\n";
    // and here, when fav goes out of scope normally
}
```

Now there is no `delete` to forget, and *every* exit path — the early `return`, the normal end of the function, even an exception thrown by `std::cout` — frees the string automatically. That is the whole point of RAII.

**14. Write a program that uses `unique_ptr::get()` to call a C-style API.**

```
#include <iostream>
#include <memory>

void c_api(int *p)
{
    *p += 1;
}

int main()
{
    auto value = std::make_unique<int>(41);

    c_api(value.get());          // hand the raw pointer to the C function

    std::cout << *value << "\n"; // prints 42
    return 0;
}
```

`.get()` returns the raw pointer that the `unique_ptr` is managing without giving up ownership. The `unique_ptr` still owns the heap integer; `c_api` only borrows it for the duration of the call.

It is critical that `c_api` does **not** call `delete` on its parameter. If it did, the `unique_ptr` would later run its own destructor and call `delete` *again* on the same address — a classic **double-free** bug, and undefined behavior. The rule for raw pointers obtained via `.get()` is "look but do not delete": treat them as observers, never as owners. If a function genuinely needs to take ownership instead, hand it the `unique_ptr` itself with `std::move`, which transfers the ownership cleanly.

# Chapter 14: Special Members and Friends

**1. Explain the difference between the Rule of Five and the Rule of Zero. Which one should you prefer and why?**

The **Rule of Five** says that if your class defines any one of the five special member functions (destructor, copy constructor, copy assignment, move constructor, move assignment), you should define all five. This is necessary when your class manages a resource directly (like raw heap memory with `new`/`delete`).

The **Rule of Zero** says that you should design your classes so that they do not need to define any special member functions. Instead, use standard library types (`std::string`, `std::vector`, `std::unique_ptr`) that manage their own resources. The compiler-generated defaults will then do the right thing.

You should prefer the **Rule of Zero** because it results in less code, fewer bugs, and classes that are easier to maintain. Only fall back to the Rule of Five when you have no choice but to manage a resource manually.

**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)) {}
};
```

The move constructor is missing `noexcept`. `std::vector` will only move elements during reallocation if the move constructor promises not to throw. Without `noexcept`, the vector falls back to copying because a failed move mid-reallocation would leave the vector in a broken state — some elements moved, others lost. The fix:

```
Track(Track &&other) noexcept
    : 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?**

`= default` tells the compiler to generate the default version of that special member function.

You need `Song() = default;` when you have already defined another constructor (like a parameterized one). Defining any constructor suppresses the compiler's automatic generation of the default constructor. Writing `= default` brings it back without you having to write the body yourself.

**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_;
};
```

The `= delete` on the copy constructor and copy assignment operator prevents `Connection` objects from being copied. Any attempt to copy a `Connection` will produce a compile-time error.

This is useful because copying a `Connection` would result in two objects managing the same file descriptor. When both are destroyed, the file descriptor would be closed twice, which is a bug. Deleting the copy operations forces the caller to use move semantics or pass by reference.

**5. Why does `operator<<` for output have to be a free function (or a friend) rather than a member function of your class?**

For `std::cout << myObject` to work, `operator<<` needs `std::ostream` as its left operand. If `operator<<` were a member function of your class, the syntax would be `myObject << std::cout`, which is backwards. The left operand of a binary operator determines which class's member function is called, and you cannot add member functions to `std::ostream` (you do not own it). So `operator<<` must be a free function, and if it needs access to private members, it must be declared as a `friend`.

**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;
}
```

It prints:

```
Vogue
```

The free function `peek` is declared as a `friend` of `Vault`, so it can access the private member `secret` directly.

**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?**

No. Friendship is not transitive. `B` being a friend of `A` means `B` can access `A`'s privates. `C` being a friend of `B` means `C` can access `B`'s privates. But that does not give `C` any access to `A`. For `C` to access `A`'s private members, `A` would need to declare `C` as a friend directly.

**8. How many of the five special member functions do you need to write?**

Zero. All members (`std::string`, `std::vector<int>`, and `int`) are types that manage themselves. The compiler-generated destructor, copy constructor, copy assignment, move constructor, and move assignment all do the right thing. This is the Rule of Zero in action.

**9. Write a class called `Album` with private members, a parameterized constructor, a `const` print function, an overloaded `==` operator, and a friend `operator<<`.**

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

class Album {
private:
    std::string title;
    std::string artist;
    int track_count;

public:
    Album(const std::string &t, const std::string &a, int tc)
```

```
        : title(t), artist(a), track_count(tc) {}

    void print() const {
        std::cout << title << " by " << artist
                  << " (" << track_count << " tracks)" << std::endl;
    }

    bool operator==(const Album &other) const {
        return title == other.title && artist == other.artist
            && track_count == other.track_count;
    }

    friend std::ostream &operator<<(std::ostream &os, const Album &a);
};

std::ostream &operator<<(std::ostream &os, const Album &a) {
    os << a.title << " by " << a.artist
       << " (" << a.track_count << " tracks)";
    return os;
}

int main()
{
    Album a("Nevermind", "Nirvana", 12);
    Album b("Tragic Kingdom", "No Doubt", 14);
    Album c("Nevermind", "Nirvana", 12);

    std::cout << a << std::endl;
    std::cout << b << std::endl;

    std::cout << (a == b) << std::endl;  // 0 (false)
    std::cout << (a == c) << std::endl;  // 1 (true)

    return 0;
}
```

**10. Write a `Buffer` with all five special members.**

```
#include <cstring>
#include <iostream>
#include <utility>

class Buffer {
    char *data;
    std::size_t len;
public:
    explicit Buffer(std::size_t n)
        : data(new char[n]), len(n)
    {
        std::cout << "default ctor (size " << len << ")\n";
    }

    ~Buffer()
    {
        std::cout << "dtor (size " << len << ")\n";
```

```
        delete[] data;
    }

    Buffer(const Buffer &other)
        : data(new char[other.len]), len(other.len)
    {
        std::memcpy(data, other.data, len);
        std::cout << "copy ctor\n";
    }

    Buffer &operator=(const Buffer &other)
    {
        std::cout << "copy assign\n";
        if (this != &other) {
            delete[] data;
            len = other.len;
            data = new char[len];
            std::memcpy(data, other.data, len);
        }
        return *this;
    }

    Buffer(Buffer &&other) noexcept
        : data(other.data), len(other.len)
    {
        other.data = nullptr;
        other.len  = 0;
        std::cout << "move ctor\n";
    }

    Buffer &operator=(Buffer &&other) noexcept
    {
        std::cout << "move assign\n";
        if (this != &other) {
            delete[] data;
            data = other.data;
            len  = other.len;
            other.data = nullptr;
            other.len  = 0;
        }
        return *this;
    }
};

int main()
{
    Buffer a(8);                // default ctor
    Buffer b = a;               // copy ctor
    Buffer c = std::move(a);    // move ctor
    return 0;                   // dtors for c, b, a (a now owns nothing)
}
```

Expected output:

```
default ctor (size 8)
```

```
copy ctor
move ctor
dtor (size 0)
dtor (size 8)
dtor (size 8)
```

Notes:

- The destructor still runs on the moved-from `a`, but its `len` is now `0` and `data` is `nullptr`, so the `delete[]` is a harmless no-op (`delete[] nullptr` is well-defined).
- Both move operations are marked `noexcept`. That is required if we want `std::vector<Buffer>` to use them during reallocation (see exercise 12).
- Both assignment operators check `this != &other` to handle the self-assignment case (see exercise 11).

**11. Where is the bug? Self-assignment.**

When the right-hand side and the left-hand side are the same object, this `operator=` first runs `delete[] data` and then tries to copy from `other.data` — but `other` *is* `*this`, so `other.data` is now a dangling pointer to memory we just freed. The subsequent loop reads from freed memory, which is undefined behavior, and the resulting `Buffer` is left in a corrupted state.

The standard fix is to detect self-assignment and bail out before doing any destructive work:

```
Buffer &operator=(const Buffer &other) {
    if (this == &other) {
        return *this;
    }
    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;
}
```

A more robust pattern is **copy-and-swap**, which gets self-assignment safety and exception safety in one step:

```
Buffer &operator=(Buffer other) {     // by value: makes a copy first
    using std::swap;
    swap(data, other.data);
    swap(len,  other.len);
    return *this;
}                                      // other's destructor frees the old buffer
```

Either form makes `b = b;` a safe no-op instead of a crash.

**12. Think about it: Why does `std::vector` insist on `noexcept` move?**

When `std::vector` runs out of capacity and has to grow, it allocates a new (larger) buffer and has to relocate every existing element from the old buffer into the new one. Vector wants this relocation to be **strongly exception safe**: if anything goes wrong partway through, the vector should be left exactly as it was before the `push_back` — the old buffer still intact, no elements lost, no half-moved state.

If the element type's move constructor is `noexcept`, the vector can move each element into the new buffer with confidence that the move cannot throw. If the move *might* throw, vector cannot recover — once you have moved 5 of 10 elements and the 6th move throws, the first 5 elements have been clobbered and there is no way to roll back. So the standard requires that vector only move elements during reallocation when their move is `noexcept`. Otherwise it falls back to **copying** them instead, which is exception-safe (if a copy throws, the originals are still untouched), but loses the entire performance benefit of move semantics.

The cost is exactly the cost of copying instead of moving. For a vector of 10,000 `std::string`s, that is 10,000

heap allocations and 10,000 character-array copies on every reallocation. For a vector of `noexcept`-movable strings, it is 10,000 pointer swaps. This is why mature classes that own resources almost always mark their move operations `noexcept`.

# Chapter 15: Odds and Ends

**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;
}
```

It prints:

```
A
```

The file does not exist, so `!f` is true. `"A"` is printed, then `exit(EXIT_FAILURE)` terminates the program immediately. Neither `"B"` nor `"C"` is ever printed.

**2. What is name mangling, and why does C++ do it but C does not?**

Name mangling is the process by which the C++ compiler encodes a function's name along with its parameter types into a unique symbol in the compiled output. For example, `void play(int)` might become `_Z4playi`.

C++ does this because it supports function overloading — multiple functions can have the same name but different parameter types. The mangled names ensure each overload has a unique symbol so the linker can tell them apart.

C does not mangle names because C does not support function overloading. Each function name is unique, so the compiler stores it as-is.

**3. A coworker gets a linker error about an undefined symbol when calling a C library function. 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;
}
```

The C++ compiler mangles the name `c_library_init` when looking for it, but the C library stored it without mangling. The fix is to declare the function with `extern "C"`:

```
extern "C" void c_library_init();
```

This tells the C++ compiler to use C-style (unmangled) naming for this function.

**4. What is the value of x after this code runs?**

```
uint8_t x = 250;
x = x + 10;
```

x is **4**.

`uint8_t` can hold values from 0 to 255. 250 + 10 = 260, which overflows. For unsigned types, overflow wraps around: 260 % 256 = 4.

**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;
}
```

It prints:

```
0
48
```

48 is the ASCII value of the character `'0'`. When printed as a `char`, it displays the character `0`. When cast to `int` and printed, it displays the numeric value `48`.

**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;
```

The C-style cast `(int*)ptr` silently converts a `void*` to an `int*` with no type checking. If `ptr` actually points to a `double`, a `std::string`, or something else entirely, you will get undefined behavior when you dereference `ip`. The C-style cast gives no indication of how dangerous this operation is.

You should use `static_cast` if you are confident about the actual type:

```
int* ip = static_cast<int*>(ptr);
```

`static_cast` is more restrictive and makes the intent clear. If you need to reinterpret the bits of one pointer type as another, use `reinterpret_cast`, which explicitly signals the danger.

**7. What is the difference between `static_cast<int>(3.14)` and `reinterpret_cast<int>(3.14)`? Will the second one even compile?**

`static_cast<int>(3.14)` performs a meaningful conversion: it converts the `double` value 3.14 to the `int` value 3 by truncating the decimal part.

`reinterpret_cast<int>(3.14)` will **not compile**. `reinterpret_cast` works on pointers and references, not on values. It reinterprets the bit pattern of one type as another, but you cannot `reinterpret_cast` a floating-point value directly to an integer value.

**8. What does the `#ifdef __cplusplus` guard accomplish in a C/C++ shared header? When would the code inside the `#ifdef` be skipped?**

The `#ifdef __cplusplus` guard wraps `extern "C" { ... }` around function declarations. When the header is compiled by a C++ compiler, `__cplusplus` is defined, so the `extern "C"` block is included, preventing name mangling. When the header is compiled by a C compiler, `__cplusplus` is not defined, so the `extern "C"` block is skipped entirely (since C does not understand `extern "C"` syntax).

This allows the same header to work correctly in both C and C++ code.

**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.**

```cpp
#include <iostream>

int main()
{
    int value = 65;
    char letter = 'Z';

    std::cout << "int " << value << " as char: "
              << static_cast<char>(value) << std::endl;
    std::cout << "char '" << letter << "' as int: "
              << static_cast<int>(letter) << std::endl;

    return 0;
}
```

Output:

```
int 65 as char: A
char 'Z' as int: 90
```

**10. What does the following program print?**

```cpp
#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;
}
```

It prints:

```
5
```

`seconds(5) + milliseconds(750)` produces a duration of 5750 milliseconds. `duration_cast<seconds>` truncates toward zero, giving 5 seconds (not 6). The fractional 750 milliseconds is discarded.

**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?**

`steady_clock` is guaranteed to never be adjusted — it always moves forward at a constant rate. `system_clock` represents the system's wall clock, which can jump forward or backward when the clock is adjusted (e.g., NTP synchronization, daylight saving time changes, or manual adjustments). If `system_clock` jumps during your measurement, you could get a negative elapsed time or an incorrectly large one. `steady_clock` avoids this problem entirely.

**12. What is wrong with this code for generating a random number between 1 and 100?**

```cpp
#include <cstdlib>
#include <iostream>

int main()
{
    int r = rand() % 100 + 1;
    std::cout << r << std::endl;

    return 0;
}
```

There are two problems:

1. `srand()` is never called, so `rand()` uses the same default seed every time the program runs, producing the same "random" number.
2. Even with `srand()`, `rand() % 100` introduces bias — if `RAND_MAX` is not evenly divisible by 100, some values are slightly more likely than others.

The proper C++ approach is to use `<random>` with `std::mt19937` and `std::uniform_int_distribution<int>(1, 100)`.

**13. Write a program that uses `<random>` to simulate rolling two six-sided dice 10 times and prints each roll.**

```cpp
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> die(1, 6);

    for (int i = 0; i < 10; ++i) {
        int d1 = die(gen);
        int d2 = die(gen);
        std::cout << "Roll " << (i + 1) << ": " << d1 << " + " << d2
                  << " = " << (d1 + d2) << std::endl;
    }

    return 0;
}
```

Sample output:

```
Roll 1: 3 + 5 = 8
Roll 2: 1 + 6 = 7
Roll 3: 4 + 4 = 8
```

```
Roll 4: 2 + 1 = 3
Roll 5: 6 + 3 = 9
Roll 6: 5 + 2 = 7
Roll 7: 1 + 4 = 5
Roll 8: 3 + 6 = 9
Roll 9: 2 + 2 = 4
Roll 10: 4 + 5 = 9
```

**14. Write a program that generates 10 random values using `std::normal_distribution` with a mean of 100 and a standard deviation of 15.**

```cpp
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::normal_distribution<double> dist(100.0, 15.0);

    for (int i = 0; i < 10; ++i) {
        std::cout << dist(gen) << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Most values will be close to 100. About 68% of values should fall between 85 and 115 (within one standard deviation of the mean), and about 95% should fall between 70 and 130 (within two standard deviations).

**15. What does the `dynamic_cast` example print?**

It prints:

```
audio
video
unknown
```

`dynamic_cast<Derived *>(base_ptr)` tries to safely down-cast a base pointer to a derived pointer. If the object actually *is* a `Derived` (or a more-derived type), the cast returns the new pointer; otherwise it returns `nullptr`. That is exactly what each `if` branch in `play` is checking. For `&a` the first `dynamic_cast` succeeds and `play` calls `AudioTrack::play()`. For `&v` the first `dynamic_cast` returns `nullptr` and the second one succeeds, so `VideoTrack::play()` runs. For the plain `Track t`, neither cast succeeds and the function falls through to `"unknown"`.

`dynamic_cast` only works when the base class has at least one `virtual` function, because it needs run-time type information stored in the object's vtable. Adding `virtual ~Track() = default;` to the base class is the minimum that gives `Track` a vtable; without it, the program would not compile (`source type is not polymorphic`). A virtual destructor is also exactly what you want anyway — without it, deleting a derived object through a `Track *` would only run `Track`'s destructor.

**16. Where is the bug? `const_cast` on a string literal-like `const std::string`.**

`const_cast` strips the `const` from a reference or pointer, allowing you to modify what was originally declared as `const`. The compiler will let you do this — but the behavior is **only** defined if the underlying object is not actually `const`.

In this program, `title` is declared `const std::string title = "wonderwall";`, so the underlying string

really is constant. When `uppercase_first` casts away the `const` and then writes through the resulting `char &`, it modifies a truly-`const` object, which is **undefined behavior**. On many compilers nothing visible happens; on others the program crashes; on still others the modification appears to "work" in debug builds but not in release builds. The fact that it "seems to work" today does not make it legal.

The right fix is not a bigger cast — it is to remove the `const` from the *original* object, or to make `uppercase_first` take its argument by non-`const` reference and let the caller pass a real mutable string:

```
void uppercase_first(std::string &s)
{
    if (!s.empty()) {
        s[0] = static_cast<char>(std::toupper(static_cast<unsigned char>(s[0])));
    }
}

int main()
{
    std::string title = "wonderwall";    // not const
    uppercase_first(title);
    std::cout << title << "\n";          // Wonderwall
    return 0;
}
```

The general rule for `const_cast`: only use it to remove `const` from something that was *not* originally declared `const`, typically when interfacing with an old C API that forgot to mark a pointer parameter `const`. Anywhere else, prefer to fix the types so the cast is unnecessary.

**17. Calculation: `<cstdint>` fixed-width types.**

| Type | Bytes | Largest value |
|---|---|---|
| `int8_t` | 1 | 127 (signed) |
| `uint8_t` | 1 | 255 (unsigned) |
| `int16_t` | 2 | 32,767 (signed) |
| `uint32_t` | 4 | 4,294,967,295 (unsigned, ~4.3 billion) |
| `int64_t` | 8 | 9,223,372,036,854,775,807 (signed) |

Use `int32_t` (or `uint32_t`, `int64_t`, etc.) when you need a *specific* width — for example, when laying out a binary file format, talking to hardware or a network protocol, or doing portable bit manipulation. The plain `int`, `long`, `long long` types are allowed to vary in size between platforms, so a struct field of type `long` is 8 bytes on Linux/macOS and 4 bytes on Windows; that variation breaks anything that depends on a specific layout.

For ordinary counters, indices, and arithmetic, prefer plain `int`. The compiler picks whatever the platform's "natural" word size is, which is usually the fastest type and avoids the noisy `int32_t`/`int64_t` spelling on every loop variable. The rule of thumb is: `int` for everyday code, `int32_t` / `int64_t` (and friends) when the width is part of the contract.

**18. Think about it: `std::exit(0)` vs `return 0;` from `main`.**

`return 0;` from `main` performs a normal function return, which means C++ first runs the destructors of all local objects in `main` (and then any globals, in reverse order of construction).

`std::exit(0)` immediately terminates the program. It does **not** unwind the stack. That means the destructors of any local objects still alive in `main` (or in any function above it on the call stack) **do not run**. Functions registered with `std::atexit` *do* run, and global / static destructors *do* run, but stack objects are skipped.

Sketch:

```
#include <cstdlib>
#include <iostream>

struct Local {
    const char *name;
    ~Local() { std::cout << "dtor " << name << "\n"; }
};

int main()
{
    Local a{"a"};
    Local b{"b"};

    if (false /* change to true to compare */) {
        std::exit(0);
    }
    return 0;
}
```

With `return 0;` the program prints:

```
dtor b
dtor a
```

(Locals are destroyed in reverse order of construction.)

With `std::exit(0)` the program prints **nothing** — both `Local` destructors are skipped. This is exactly why `std::exit` is a sledgehammer: anything an RAII object was holding open (a file, a database connection, a temporary directory) is left dangling. Prefer `return` from `main` whenever possible, and reserve `std::exit` for cases where you want to abort early from deep inside a call chain *and* you have already cleaned up anything that needed cleaning up.

**19. Write a program that seeds `std::mt19937` from `std::random_device`.**

```
#include <iostream>
#include <random>

int main()
{
    std::random_device rd;
    std::mt19937 rng(rd());
    std::uniform_int_distribution<int> dist(1, 100);

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

`std::random_device` is a hardware-backed source of entropy where available; on systems without one, the standard library still provides a `std::random_device` that returns *some* unpredictable bits at startup. Either way, two runs of this program produce **different** sequences of numbers, because each run reads a fresh seed from `random_device`.

If you replace the seeding line with a fixed constant:

```
std::mt19937 rng(42);
```

then the engine starts in exactly the same state every run, and **the two runs print exactly the same 10 numbers**. That is a feature, not a bug: it makes randomized programs reproducible (useful for tests and for debugging a problem you only see "sometimes") at the cost of being predictable. For anything where unpredictability matters (games, simulations, anything user-facing), seed from `random_device`; for tests and reproducible experiments, seed from a known constant.