https://simplifycpp.org

Modern C++

From Zero to Professional

A Complete Guide to Learning Modern C++ from Fundamentals to Advanced Topics

DRAFT

Prepared by Ayman Alheraki

First Edition

Modern C++ From Zero to Professional

A Complete Guide to Learning Modern C++ from Fundamentals to Advanced Topics

Some drafting assistance and idea exploration were supported by modern AI tools, with full supervision, verification, correction, and authorship.

Prepared by Ayman Alheraki

March 2026

Contents

Author's Introduction

Over the course of my professional journey in software engineering, I have written and prepared a large number of technical books, guides, and educational materials. Most of these works were highly specialized and targeted toward advanced domains, including compilers, low-level toolchains, modern C++ internals, system design, and performance-oriented programming.
These specialized works were written with a clear purpose: to address deep technical topics that require precision, experience, and a strong background in systems programming. They were intended for readers who had already crossed the foundational stages and were ready to explore the deeper layers of programming languages, architecture, and software engineering.
However, over time, I came to recognize an important and recurring challenge.
Many learnerswhether beginners or even intermediate developersstruggle not because the topics themselves are impossible, but because the path toward them is fragmented. They often encounter advanced materials too early, or they rely on beginner resources that stop before building a solid and professional foundation. As a result, their knowledge remains incomplete, disconnected, or dependent on memorization rather than understanding.
This realization led me to a crucial conclusion: there is a strong need for a book that does not specialize too early, but instead builds the foundation correctly from the beginning and continues all the way to the advanced level in a single, coherent journey.
This book is the result of that conclusion.
It is the first book I have designed specifically to focus on the fundamentals of C++ in a detailed, structured, and comprehensive way, while still maintaining a clear path toward professional-level expertise. Unlike my previous works, which assumed prior experience and focused on depth within specific domains, this book starts from zero and builds upward carefully, without skipping essential concepts.
The objective is not to simplify the language in a superficial way, but to make it accessible without losing its true nature. C++ is a language that rewards depth, precision, and discipline. It cannot be mastered through shortcuts or fragmented learning.
Therefore, this book respects the complexity of C++, but presents it in a way that is progressive, logical, and approachable.
One of the key motivations behind writing this book is the desire to provide what I believe many learners truly need: a clear and reliable starting point, followed by a guided progression that does not leave gaps. Every concept introduced here is placed intentionally within a broader structure. The reader is not expected to jump between unrelated topics or search for missing explanations elsewhere.
This book is also influenced by long years of practical experience in real-world software development. It is shaped by building systems, solving problems, writing large codebases, maintaining software over time, and understanding what truly matters when writing code that must work reliably and efficiently.
Because of that, the content is not purely academic. It is grounded in practice.
The examples, explanations, and design choices throughout the book aim to reflect real engineering thinking. The reader will not only learn how features work, but also when they should be used, why they exist, and what trade-offs they involve. This is essential for transitioning from a learner to a professional.
Another important aspect of this work is its modern perspective.

C++ has evolved significantly over the past decade. The language today is not the same as it was in earlier eras. Modern C++ introduces better tools for safety, abstraction, expressiveness, and performance. Concepts such as RAII, move semantics, smart pointers, type inference, generic programming, and structured abstractions are no longer optional knowledgethey are fundamental to writing correct and maintainable code.

For that reason, this book does not follow a legacy-first approach. It does not train the reader in outdated patterns and then attempt to correct them later. Instead, it introduces modern practices from the beginning, allowing the learner to build correct habits early and avoid unnecessary confusion.

This approach is especially important for beginners. The first concepts a programmer learns often shape their long-term habits. By presenting modern C++ as the default, this book helps ensure that the reader develops a mindset aligned with current professional standards.

At the same time, this book does not assume that learning will be easy.

C++ is a powerful language, and with that power comes complexity. Understanding object lifetimes, value categories, templates, memory management, and concurrency requires effort and patience. This book does not attempt to hide that reality. Instead, it provides a structured path that makes the complexity manageable and meaningful.

Every chapter is designed to build on what came before. The reader is guided step by step, with each concept reinforcing previous knowledge while introducing new ideas. This gradual progression is essential for deep understanding.

It is also important to emphasize that this book is not intended to be read passively.

Learning C++ requires active engagement. The reader is strongly encouraged to compile every example, experiment with modifications, explore edge cases, and observe the behavior of the code. This process transforms theoretical understanding into practical skill.

Over time, this active approach builds confidence. What initially seems complex becomes familiar. What once required effort becomes natural. This transformation is the true goal of the learning journey.

This book is also meant to serve as a long-term reference.

It is not limited to a single reading. As the reader progresses in their career, they may return to different sections to revisit concepts, clarify details, or refine their understanding. The structure of the book supports this kind of repeated use.

Finally, this work reflects a broader belief about programming and learning.

Programming languages are not just tools; they are ways of thinking. C++ in particular teaches precision, responsibility, and awareness of how software interacts with the underlying system. It encourages the developer to think carefully about design, performance, and correctness.

By learning C++ deeply, a programmer does not only gain the ability to write codethey gain a stronger understanding of how software systems are built and how they behave.

This is the value that this book aims to provide.

It is my hope that this work will serve as a reliable guide for anyone who is serious about learning Modern C++, whether they are starting from zero or seeking to refine and strengthen their existing knowledge.

The journey from beginner to professional is not short, and it is not always easy. But with the right structure, the right mindset, and consistent effort, it is entirely achievable.

This book is designed to walk that journey with you, from the very first step to a level of confidence and capability that defines a true C++ professional.

Ayman Alheraki

Preface

Modern C++ is no longer simply an extension of the C language. It has evolved into a powerful, expressive, and multi-paradigm programming language that enables developers to build everything from low-level system components to high-performance distributed backend systems and large-scale applications. This book, **Modern C++ From Zero to Professional**, is designed to guide the reader through this evolution step by step, starting from the absolute fundamentals and progressing toward the most advanced features available in the latest standards.

C++ has undergone continuous transformation through successive standards: C++11 introduced a new era of expressiveness and safety, followed by incremental improvements in C++14 and C++17, and then major paradigm-shifting features in C++20 such as concepts, modules, coroutines, and ranges.

More recently, C++23 has further refined the language and standard library, improving usability, performance, and developer productivity. This book is written with a modern mindset, focusing on current and forward-looking C++ practices rather than legacy techniques that have been replaced or discouraged.

The primary goal of this book is to provide a complete, structured, and practical learning path that transforms a beginner into a professional-level C++ developer. Unlike traditional books that either focus only on syntax or isolate advanced topics, this guide integrates all essential aspects of C++ into a unified learning experience:

- Core language fundamentals and syntax
- Memory management and resource ownership (RAII)
- Object-oriented and generic programming
- The Standard Library and STL components
- Concurrency and multithreading
- Modern abstractions such as concepts and ranges
- System-level understanding and performance considerations

The design philosophy of this book is strongly influenced by the principles of modern C++ guidelines: writing code that is safer, clearer, and more maintainable while preserving performance.

Every concept introduced in this book is presented with a clear rationale, practical examples, and real-world usage patterns to ensure deep understanding rather than superficial familiarity.

This book assumes no prior knowledge of C++, but it does assume seriousness and commitment from the reader. Learning C++ is not about memorizing syntax; it is about understanding how software interacts with memory, the compiler, and the underlying hardware. This is why the journey begins with foundational concepts such as compilation, data types, and control

structures, and gradually expands toward advanced topics like template metaprogramming, concurrency models, and modern architectural patterns.
All examples in this book are written to be compiled and executed on a Windows environment using modern compilers such as MSVC, Clang, or GCC. The reader is encouraged to actively experiment with the provided code, modify it, and explore its behavior. Practical engagement is essential for mastering C++.

What Makes This Book Different

This book differs from many existing resources in several important ways:

- **Modern-first approach:** Only modern C++ practices are taught. Deprecated or outdated techniques are avoided unless necessary for understanding evolution.
- **End-to-end coverage:** The content spans from zero knowledge to advanced professional-level topics.
- **Practical orientation:** Every concept is supported by real, compilable examples.
- **System-level perspective:** The reader gains insight into how C++ interacts with compilers, memory, and hardware.
- **Professional mindset:** The focus is not only on writing code, but on writing high-quality, maintainable, and efficient systems.

How to Use This Book

To get the maximum benefit from this book:

- Follow the chapters sequentially, especially if you are a beginner.
- Compile and run every example provided.
- Experiment with modifications to understand behavior deeply.
- Revisit advanced chapters multiple times as your understanding grows.

Each chapter builds upon the previous ones, forming a coherent and progressive learning path. Skipping foundational topics may lead to gaps in understanding when dealing with advanced material.

Philosophy of Learning C++

C++ is a language that rewards depth. It allows the programmer to operate at multiple levels of abstraction, from high-level design to low-level memory manipulation. The Standard Library, which evolved from the original STL, provides powerful generic components such as containers, algorithms, iterators, and function objects that enable efficient and reusable code. However, mastering C++ requires more than knowing its features. It requires understanding when and why to use them. This book emphasizes:

- Writing expressive and readable code

- Avoiding unnecessary complexity
- Leveraging zero-cost abstractions
- Designing systems with performance in mind

Final Words

C++ remains one of the most powerful and relevant programming languages in the world. It is used in operating systems, game engines, financial systems, embedded devices, and high-performance computing. Its ability to combine abstraction with efficiency makes it unique among modern programming languages.

This book is not just a guide to learning a programming language; it is a guide to thinking like a professional systems developer. By the end of this journey, the reader will not only understand Modern C++ but will also be capable of designing and building robust, efficient, and scalable software systems.

The journey from zero to professional is not easybut it is one of the most rewarding paths in software engineering.

Part I

Getting Started the Right Way

What is Modern C++?

Modern C++ is the modern form of the C++ programming language shaped by standards from C++11 onward. It represents a shift from older styles toward safer, clearer, and more expressive programming while maintaining the core strength of performance and control.

1.1 Why C++ is still one of the most powerful languages

C++ remains one of the most powerful programming languages because it uniquely combines low-level control with high-level abstraction.
It allows:

- Direct control over memory and hardware
- High-level abstractions with minimal overhead
- Multi-paradigm programming (procedural, OOP, generic, functional)
- High performance comparable to C
- A rich and evolving standard library

Example:

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

int main() {
    std::vector<int> values{1,2,3,4,5};

    for (int v : values) {
        std::cout << v << '\n';
    }
}
```

This demonstrates abstraction without losing efficiency.

1.2 Difference between C and C++

C and C++ share a common history but differ significantly in design philosophy.
C focuses on:

- Procedural programming
- Manual memory management
- Minimal abstraction

C++ provides:

- Classes and object modeling
- RAII-based resource management
- Templates and generic programming
- Standard containers and algorithms

Example comparison:
C-style:

```
int arr[3] = {1,2,3};
```

C++ style:

```
#include <vector>
std::vector<int> arr{1,2,3};
```

1.3 Old C++ vs Modern C++

Old C++ relied heavily on:

- Raw pointers
- Manual memory management
- C-style arrays

Modern C++ emphasizes:

- RAII
- Smart pointers
- STL containers
- Move semantics
- Safer abstractions

Old style:

```
int* p = new int(5);
delete p;
```

Modern style:

```
#include <memory>
auto p = std::make_unique<int>(5);
```

1.4 Zero-overhead philosophy

C++ follows a key principle:

- What you do not use, you do not pay for
- What you use is as efficient as hand-written code

Example:

```
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> v{1,2,3,4};

    auto count = std::count_if(v.begin(), v.end(),
        [](int x){ return x % 2 == 0; });
}
```

This abstraction compiles efficiently without hidden overhead.

1.5 Why you should not start with C-style programming in C++

Starting with C-style code leads to bad habits:

- Using raw pointers unnecessarily
- Avoiding standard library features
- Manual memory management errors

Bad example:

```
char* str = (char*)malloc(100);
```

Correct modern approach:

```
#include <string>
std::string str = "Modern C++";
```

Modern C++ should be learned as its own language, not as extended C.

1.6 Evolution from C++11 to C++23 and the direction toward C++26

C++ has evolved significantly:

1.6.1 C++11

- auto, lambda, move semantics
- smart pointers
- threading support

1.6.2 C++17

- structured bindings
- std::optional, std::variant
- filesystem

1.6.3 C++20

- concepts
- ranges
- coroutines
- modules

1.6.4 C++23

- std::expected
- improved ranges
- usability improvements

Example (Concepts):

```
#include <concepts>

template <std::integral T>
T add(T a, T b) {
    return a + b;
}
```

1.6.5 Direction toward C++26

Future direction includes:

- Better reflection support
- Improved compile-time capabilities
- Cleaner and safer language rules

Conclusion

Modern C++ is not just an updated version of C++. It is a complete, powerful, and evolving system programming language.
To learn it correctly, one must start with modern principles from the beginning, not legacy practices.
This book follows that path.

Setting Up the Development Environment

A strong beginning in Modern C++ depends not only on learning the language itself, but also on preparing a correct and reliable development environment. Many beginner problems do not come from the language rules alone. They come from using the wrong compiler mode, misunderstanding how files are built, or not knowing whether an error comes from compilation or linking.
This chapter establishes the practical foundation for all the code that follows in this book. It explains how to choose a compiler, how to select an appropriate language standard, how to write and build a first Modern C++ program, how to compile from the command line, how to prepare commonly used development environments on Windows, and how to understand the essential stages of building a program.
The goal is not simply to make programs compile. The goal is to help the reader understand what the compiler, the build system, and the linker are doing, because that understanding becomes increasingly important as projects grow.

2.1 Choosing a compiler: GCC / Clang / MSVC

A C++ compiler translates C++ source code into object code and, together with a linker and the surrounding toolchain, helps produce the final executable program. On Windows, the three most commonly discussed compiler families are GCC, Clang, and MSVC.
A beginner should understand early that these compilers all aim to implement the C++ language, but they differ in platform integration, diagnostics, tooling, ecosystem, and practical workflow.

2.1.1 GCC

GCC, the GNU Compiler Collection, is one of the most established compiler families in the C and C++ world. On Windows, GCC is commonly used through environments such as MinGW-w64. GCC is widely respected for portability, broad platform support, mature optimization capabilities, and long-term presence in native software development.
In practice, many cross-platform developers use GCC because:

- it is common in open-source projects,
- it has broad support for ISO C++ modes,
- it works well with command-line and script-based workflows,
- it is often paired with tools such as GDB, Make, and CMake.

A typical GCC compiler driver for C++ is `g++`.
Example:

```
g++ -std=c++20 -Wall -Wextra -pedantic main.cpp -o app.exe
```

This command compiles `main.cpp` using the C++20 language mode and produces an executable named `app.exe`.

2.1.2 Clang

Clang is part of the LLVM project. It is known for clean architecture, strong diagnostics, fast parsing, excellent tooling integration, and widespread use in advanced development environments. Clang is especially appreciated for high-quality error messages and for its ecosystem value in static analysis, language tooling, formatting, and refactoring.
On Windows, Clang may be used in different ways:

- as `clang++` with its own toolchain,
- with GNU-like command-line usage,
- or in configurations that interoperate with Microsoft tooling.

Example:

```
clang++ -std=c++20 -Wall -Wextra -pedantic main.cpp -o app.exe
```

For many learners, Clang is an excellent choice when they want modern diagnostics and a toolchain style similar to GCC.

2.1.3 MSVC

MSVC, the Microsoft Visual C++ compiler, is the native Microsoft compiler family for Windows development. It is tightly integrated with Visual Studio, the Windows SDK, the Microsoft debugger, the Visual Studio Developer Command Prompt, and the surrounding Windows-native development workflow.
MSVC is especially important on Windows because:

- it is the most deeply integrated native compiler in the Visual Studio ecosystem,
- it works directly with the Microsoft project system,
- it is widely used in professional Windows development,
- it offers current ISO C++ standard modes through the `/std` option.

A simple MSVC command-line example:

```
cl /EHsc /std:c++20 main.cpp
```

This compiles `main.cpp` in C++20 mode and, by default, creates an executable with a name derived from the source file.

2.1.4 Which compiler should a beginner choose?

For a Windows-based learner, the practical recommendation is usually:

- choose **MSVC** if you want the smoothest native Windows integration and intend to use Visual Studio heavily,
- choose **Clang** if you value diagnostics and modern LLVM-based tooling,
- choose **GCC** if you want strong compatibility with common GNU-style build workflows or open-source conventions.

A very good professional habit is to test code with more than one compiler when possible. Different compilers may diagnose different issues, warn about different patterns, or expose portability assumptions.

2.1.5 A practical comparison example

The following source file is valid across GCC, Clang, and MSVC:

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

int main() {
    std::vector<int> values{1, 2, 3, 4, 5};

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

This same source can be compiled with all three compiler families:

```
g++ -std=c++20 main.cpp -o app.exe
clang++ -std=c++20 main.cpp -o app.exe
```

```
cl /EHsc /std:c++20 main.cpp
```

The source language is the same, but the tool invocation differs.

2.2 Selecting the correct language standard

One of the most important setup decisions in C++ is selecting the intended language standard. A C++ compiler does not automatically behave as “the latest C++” in every configuration. The active language mode affects what syntax is accepted, what library facilities are available, and which behaviors follow which standard rules.

2.2.1 Why the language standard matters

Modern C++ evolved significantly from C++11 through C++23, with ongoing work toward C++26. If you compile a program without selecting the intended standard mode, you may see errors for perfectly correct modern code.
For example, the following uses structured bindings, which require C++17 or later:

```cpp
#include <tuple>
#include <iostream>

int main() {
    std::tuple<int, double> data{7, 3.5};
    auto [count, ratio] = data;

    std::cout << count << ' ' << ratio << '\n';
}
```

If this source is compiled in an older standard mode, it may fail.

2.2.2 Common standard modes

The most important modern modes for learners are:

- `C++17`
- `C++20`
- `C++23`

For most new learning and new projects, **C++20** is an excellent default because it is modern, widely supported, and includes major language improvements such as concepts and ranges. Where toolchain support is stable enough for the specific environment, **C++23** is also a strong option for experimentation and gradual adoption.

2.2.3 Selecting the standard with GCC

Examples:

```
g++ -std=c++17 main.cpp -o app.exe
g++ -std=c++20 main.cpp -o app.exe
g++ -std=c++23 main.cpp -o app.exe
```

GNU dialect variants also exist:

```
g++ -std=gnu++20 main.cpp -o app.exe
```

The difference is that `gnu++20` enables GNU extensions in addition to the ISO language mode, while `c++20` aims at the ISO mode itself.

2.2.4 Selecting the standard with Clang

Examples:

```
clang++ -std=c++17 main.cpp -o app.exe
clang++ -std=c++20 main.cpp -o app.exe
clang++ -std=c++23 main.cpp -o app.exe
```

2.2.5 Selecting the standard with MSVC

Examples:

```
cl /EHsc /std:c++17 main.cpp
cl /EHsc /std:c++20 main.cpp
cl /EHsc /std:c++latest main.cpp
```

In Microsoft environments, `/std:c++latest` enables the latest supported language and library features that are available in that compiler version. Depending on the installed Visual Studio version, preview support for newer standards may also exist through preview-oriented settings.

2.2.6 A practical recommendation

For the code in this book, a good learning default on Windows is:

- use `C++20` as the main standard,
- test `C++23` when your installed compiler supports the needed features,
- avoid older modes unless a chapter specifically discusses historical comparison or compatibility.

2.3 Your first modern C++ program

A first Modern C++ program should already reflect good habits. It should not be written in a purely C-style form if the goal is to learn Modern C++ correctly from the beginning.
A simple and correct first example is:

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

int main() {
    std::string message = "Welcome to Modern C++";
    std::cout << message << '\n';
    return 0;
}
```

This short program already introduces several foundational ideas:

- a header is included with `#include`,
- `main` is the program entry point,
- `std::string` is preferred over a raw C-style character buffer for ordinary text handling,
- `std::cout` writes to standard output,
- the program returns an integer status code.

2.3.1 Why this is already Modern C++

Although very small, this program is better than older C-style introductions because it encourages:

- using standard library types immediately,
- thinking in terms of typed objects rather than raw memory,
- writing expressive code with safe default facilities.

A slightly richer first example can also demonstrate containers:

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

int main() {
    std::vector<int> numbers{10, 20, 30, 40};

    for (int value : numbers) {
        std::cout << value << '\n';
    }
}
```

This introduces `std::vector` and range-based `for`, both of which are core parts of normal modern practice.

2.4 Compiling from the command line

A programmer who understands command-line compilation gains an important advantage: they begin to see the build process directly rather than as a hidden IDE action. Even when using Visual Studio, VS Code, or CLion, understanding the command-line model helps explain what the environment is actually doing.

2.4.1 Compiling with GCC

Suppose the source file is named `main.cpp`. A typical command is:

```
g++ -std=c++20 -Wall -Wextra -pedantic main.cpp -o app.exe
```

This means:

- use the GNU C++ compiler driver,
- compile in C++20 mode,
- enable common warnings,
- produce the executable `app.exe`.

2.4.2 Compiling with Clang

```
clang++ -std=c++20 -Wall -Wextra -pedantic main.cpp -o app.exe
```

This is conceptually similar to GCC.

2.4.3 Compiling with MSVC

In the Visual Studio Developer Command Prompt:

```
cl /EHsc /std:c++20 /W4 main.cpp
```

This means:

- use the Microsoft compiler,

- enable standard C++ exception handling assumptions,
- compile in C++20 mode,
- use warning level 4.

2.4.4 Running the executable

After successful compilation:

```
app.exe
```

2.4.5 Compiling multiple source files

Real projects often contain more than one source file. For example:

```
// math_utils.h
#pragma once

int add(int a, int b);
```

```
// math_utils.cpp
#include "math_utils.h"

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

```
// main.cpp
#include <iostream>
#include "math_utils.h"

int main() {
    std::cout << add(10, 20) << '\n';
}
```

Compile with GCC or Clang:

```
g++ -std=c++20 main.cpp math_utils.cpp -o app.exe
clang++ -std=c++20 main.cpp math_utils.cpp -o app.exe
```

Compile with MSVC:

```
cl /EHsc /std:c++20 main.cpp math_utils.cpp
```

2.4.6 Separate compilation

A more professional workflow separates compilation and linking.
With GCC or Clang:

```
g++ -std=c++20 -c math_utils.cpp -o math_utils.o
g++ -std=c++20 -c main.cpp -o main.o
g++ main.o math_utils.o -o app.exe
```

With MSVC:

```
cl /EHsc /std:c++20 /c math_utils.cpp
cl /EHsc /std:c++20 /c main.cpp
link main.obj math_utils.obj
```

This makes the build stages easier to understand.

2.5 Setting up Visual Studio / VS Code / CLion

A development environment combines the editor or IDE, the compiler toolchain, and supporting tools such as the debugger and build system. On Windows, three practical environments are especially common for C++ learners.

2.5.1 Setting up Visual Studio

Visual Studio is the most integrated Windows-native C++ environment.
A good setup approach is:

- install Visual Studio 2022,
- select the `Desktop development with C++` workload,
- ensure the MSVC toolset and Windows SDK are installed,
- create a Console App project for first experiments,
- set the language standard to C++20 or a newer supported mode.

A typical workflow:

1. Open Visual Studio.
2. Create a new project.
3. Choose `Console App`.
4. Select a project name and location.
5. Open project properties.
6. Under C/C++ language settings, choose the intended C++ language standard.
7. Build and run the project.

A simple source file inside a Visual Studio project:

```cpp
#include <iostream>

int main() {
    std::cout << "Visual Studio C++ environment ready\n";
}
```

2.5.2 Using the Developer Command Prompt

Visual Studio also installs a developer command environment that configures the required compiler, linker, include paths, and library paths.
Example:

```
cl /EHsc /std:c++20 hello.cpp
```

This command works correctly because the prompt has already been initialized for the MSVC toolchain.

2.5.3 Setting up VS Code

VS Code is a lightweight editor rather than a full IDE by default, but it becomes a strong C++ environment when paired with the proper extensions and compiler toolchain.
A practical Windows setup is:

- install VS Code,
- install the Microsoft C/C++ extension,
- install a compiler toolchain such as MSVC, MinGW-w64 GCC, or Clang,
- configure build and debug tasks.

A common beginner workflow with MSVC is:

1. install Visual Studio Build Tools or full Visual Studio,
2. open VS Code,
3. install the C/C++ extension,
4. create a folder for the project,
5. create `main.cpp`,
6. configure the compiler path and IntelliSense,
7. create a build task and run it.

Example source:

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

int main() {
    std::vector<int> values{2, 4, 6, 8};

    for (int v : values) {
        std::cout << v << ' ';
    }

    std::cout << '\n';
}
```

A representative VS Code task using MinGW-w64 GCC might invoke:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build with g++",
            "type": "shell",
            "command": "g++",
            "args": [
                "-std=c++20",
                "-Wall",
                "-Wextra",
                "-pedantic",
                "main.cpp",
                "-o",
                "app.exe"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

2.5.4 Setting up CLion

CLion is a full C and C++ IDE centered strongly around CMake. It is especially attractive for learners who want an integrated modern environment with code navigation, refactoring support, debugging, and clean project structure.
A practical Windows setup is:

- install CLion,
- ensure a supported compiler toolchain is available,
- create or open a CMake project,
- configure the selected toolchain,
- build and run through the IDE.

A minimal `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.20)

project(ModernCppStart LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

add_executable(ModernCppStart main.cpp)
```

And a matching `main.cpp`:

```cpp
#include <iostream>

int main() {
    std::cout << "CLion environment ready\n";
}
```

2.5.5 Which environment should you use?

A practical recommendation is:

- use **Visual Studio** if you want the strongest native Windows integration,
- use **VS Code** if you prefer a lightweight editor and flexible toolchain setup,
- use **CLion** if you want an integrated CMake-centered workflow.

All three are valid. What matters is that the environment is configured correctly and that the learner understands the compiler and build steps underneath it.

2.6 Source files vs header files

C++ projects are commonly divided into source files and header files. This convention is essential for organizing declarations, definitions, interfaces, and implementation units.

2.6.1 Header files

Header files usually contain declarations that other files need to know about.
Common header contents include:

- function declarations,
- class declarations,
- struct definitions,
- constants,
- inline functions,
- templates,
- declarations of external variables.

Example:

```cpp
// printer.h
#pragma once

void print_message();
```

2.6.2 Source files

Source files usually contain definitions and implementation details.
Example:

```
// printer.cpp
#include <iostream>
#include "printer.h"

void print_message() {
    std::cout << "Printed from printer.cpp\n";
}
```

The source that uses it:

```
// main.cpp
#include "printer.h"

int main() {
    print_message();
}
```

2.6.3 Why this separation matters

Separating headers from source files helps:

- expose interfaces without exposing implementation details,
- reduce duplication,
- organize larger projects,
- support separate compilation.

2.6.4 Declarations and definitions

A declaration introduces a name and its type or interface.
A definition provides the full implementation or storage.
Example:

```
// declaration
int add(int a, int b);

// definition
int add(int a, int b) {
    return a + b;
}
```

2.6.5 Header guards and #pragma once

Headers are often included in multiple files. To prevent repeated inclusion problems, a header should be protected.
Two common techniques are:

```
#pragma once
```

or classic include guards:

```
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);

#endif
```

2.7 Understanding build, compile, and link

Many beginners use the word “compile” to mean the whole process of producing an executable. In practice, building a C++ program usually involves several stages.

2.7.1 Preprocessing

Before compilation proper, the preprocessor handles directives such as:

- `#include`
- `#define`
- conditional compilation directives

Example:

```
#include <iostream>
#define APP_NAME "Modern C++"

int main() {
    std::cout << APP_NAME << '\n';
}
```

The preprocessor expands the macro and inserts included file contents before compilation continues.

2.7.2 Compilation

The compiler translates a translation unit into object code. A translation unit is essentially a source file after preprocessing.
Example:

```
g++ -std=c++20 -c main.cpp -o main.o
```

This does not create the final executable. It creates an object file.

2.7.3 Linking

The linker combines object files and libraries into the final executable.
Example:

```
g++ main.o math_utils.o -o app.exe
```

If all referenced symbols are found, the linker produces the executable.

2.7.4 Build

The word **build** usually refers to the full process:

- preprocessing,
- compiling,
- assembling if applicable within the toolchain,
- linking,
- and sometimes additional project steps.

2.7.5 A full mental model

The following project:

```
// math_utils.h
#pragma once

int multiply(int a, int b);
```

```
// math_utils.cpp
#include "math_utils.h"

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

```
// main.cpp
#include <iostream>
#include "math_utils.h"

int main() {
    std::cout << multiply(6, 7) << '\n';
}
```

Can be understood as:

1. the preprocessor expands `#include`,
2. each `.cpp` file is compiled into an object file,
3. the linker resolves the reference to `multiply`,
4. the final executable is produced.

2.8 Compiler errors vs linker errors

A beginner must learn to distinguish compiler errors from linker errors. This saves enormous time and confusion.

2.8.1 Compiler errors

Compiler errors happen when the compiler cannot successfully translate a source file.
Typical causes include:

- syntax errors,
- undeclared names,
- type mismatches,
- invalid use of language features,
- missing required headers.

Example of a compiler error:

```
#include <iostream>

int main() {
    std::cout << "Hello"
}
```

This is missing a semicolon after the output statement, so the compiler reports a syntax error.
Another compiler error:

```
int main() {
    value = 10;
}
```

Here `value` was never declared, so compilation fails.

2.8.2 Linker errors

Linker errors happen later. Compilation may succeed for each source file, but linking fails because the final program refers to symbols whose definitions are missing or inconsistent.
Example:

```
// math_utils.h
#pragma once

int subtract(int a, int b);
```

```
// main.cpp
#include <iostream>
#include "math_utils.h"

int main() {
    std::cout << subtract(10, 3) << '\n';
}
```

If `subtract` is declared but no definition is provided anywhere, compilation of `main.cpp` may succeed, but linking fails because the program needs a real definition.

2.8.3 A typical unresolved external symbol situation

This example creates a classic linker failure:

```
// math_utils.cpp
int add(int a, int b) {
    return a + b;
}
```

```
// main.cpp
#include <iostream>

int add(int a, int b);

int main() {
    std::cout << add(1, 2) << '\n';
}
```

If only `main.cpp` is compiled and linked, but `math_utils.cpp` is forgotten, the compiler can compile `main.cpp`, yet the linker will fail because the function body was never linked into the final program.
Correct build:

```
g++ -std=c++20 main.cpp math_utils.cpp -o app.exe
```

or

```
cl /EHsc /std:c++20 main.cpp math_utils.cpp
```

2.8.4 How to think about the difference

A very practical rule is:

- if the source code itself is invalid, think **compiler error**,
- if the code compiles but the program cannot be assembled into a final executable, think **linker error**.

2.8.5 Another common linker case: multiple definitions

Linking can also fail if the same non-inline definition appears in more than one translation unit.
Problematic example:

```
// file1.cpp
int global_value = 10;
```

```
// file2.cpp
int global_value = 20;
```

These produce conflicting definitions during linking.
A better design usually places a single definition in one source file and uses a declaration elsewhere when necessary.

Conclusion

A correct development environment is one of the most important foundations for learning C++ professionally. The programmer must choose a suitable compiler, select the intended language standard explicitly, understand how to compile simple and multi-file programs, and learn to distinguish between source-level problems and link-time problems.
GCC, Clang, and MSVC are all capable and important compilers, but their workflows differ. Visual Studio, VS Code, and CLion each provide a valid Windows-based environment when configured correctly. Source files and header files serve different organizational purposes, and the path from source code to executable passes through distinct build stages that every serious C++ programmer should understand.
The deeper lesson of this chapter is simple: C++ development should not feel mysterious. The more clearly you understand the environment, the easier it becomes to learn the language itself.
In the next chapters, that environment becomes the foundation on which all real C++ learning will be built.

Structure of a C++ Program

Understanding the structure of a C++ program is the first essential step toward writing correct and maintainable code. Before learning advanced features, a programmer must understand how a program is organized, how execution begins, how code is grouped, and how different components interact.
A C++ program is not just a sequence of instructions. It is a structured system composed of translation units, declarations, definitions, scopes, and logical groupings that together form a complete executable.
This chapter introduces the fundamental building blocks of every C++ program.

3.1 The `main` function

Every C++ program begins execution from a function named `main`. This function is the entry point of the program, and the operating system calls it when the program starts.
The most common forms of `main` are:

```
int main() {
    return 0;
}
```

or with command-line arguments:

```
int main(int argc, char* argv[]) {
    return 0;
}
```

3.1.1 Return value of `main`

The return value of `main` indicates the program's exit status:

- `0` typically indicates success
- non-zero values indicate errors

In modern C++, reaching the end of `main` without a `return` statement implicitly returns `0`.

3.1.2 Example

```
#include <iostream>

int main() {
    std::cout << "Program started\n";
}
```

3.2 Statements and comments

A C++ program is composed of statements. A statement is a complete instruction that the compiler can execute.

3.2.1 Types of statements

Common types include:

- expression statements
- declaration statements
- control statements
- compound statements (blocks)

Example:

```
int x = 10;        // declaration statement
x = x + 5;         // expression statement

if (x > 10) {      // control statement
    x = 0;
}
```

3.2.2 Comments

Comments are ignored by the compiler and are used to document code.
Single-line comment:

```
// This is a single-line comment
```

Multi-line comment:

```
/*
   This is a multi-line comment
*/
```

3.2.3 Good use of comments

Comments should explain **why**, not repeat what the code already clearly expresses.

3.3 Code formatting and style

Code formatting is essential for readability, maintainability, and collaboration. Although the compiler ignores formatting, humans depend on it.

3.3.1 Basic formatting principles

- consistent indentation
- clear spacing
- meaningful naming
- one statement per line

Poor formatting:

```c
int main(){int x=10;if(x>5){x++;}}
```

Good formatting:

```c
int main() {
    int x = 10;

    if (x > 5) {
        x++;
    }
}
```

3.3.2 Brace styles

Different styles exist (K&R, Allman), but consistency is more important than the specific choice.

3.3.3 Naming conventions

Common practices include:

- variables: `lower_case`
- types/classes: `PascalCase`
- constants: `UPPER_CASE`

3.4 Scopes { }

Scope defines where a name is visible and accessible.

3.4.1 Block scope

A block is defined by curly braces:

```c
{
    int x = 10;
}
```

Here, `x` exists only inside the block.

3.4.2 Nested scopes

```
int main() {
    int x = 5;

    {
        int x = 10; // inner scope
    }

    return 0;
}
```

The inner x hides the outer one within that scope.

3.4.3 Lifetime vs scope

Scope determines visibility, while lifetime determines how long an object exists.

3.5 Declarations vs definitions

Understanding the difference between declaration and definition is fundamental.

3.5.1 Declaration

A declaration introduces a name and its type.

```
int add(int a, int b); // declaration
```

3.5.2 Definition

A definition provides the actual implementation or storage.

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

3.5.3 Variables

```
extern int value; // declaration
int value = 10;   // definition
```

3.5.4 Why the distinction matters

Separating declarations and definitions enables:

- multi-file projects
- faster compilation
- better organization

3.6 .cpp and .hpp files

C++ programs are typically divided into source files and header files.

3.6.1 Source files (.cpp)

Contain implementations:

```
// math.cpp
int add(int a, int b) {
    return a + b;
}
```

3.6.2 Header files (.hpp)

Contain declarations:

```
// math.hpp
#pragma once

int add(int a, int b);
```

3.6.3 Usage

```
// main.cpp
#include <iostream>
#include "math.hpp"

int main() {
    std::cout << add(2, 3) << '\n';
}
```

3.6.4 Why separation is important

- improves modularity
- avoids duplication
- supports large projects

3.7 Introduction to namespaces

Namespaces are used to organize code and avoid name conflicts.

3.7.1 Basic namespace

```
namespace math {
    int add(int a, int b) {
        return a + b;
    }
}
```

3.7.2 Using namespace members

```
#include <iostream>

namespace math {
    int add(int a, int b) {
        return a + b;
    }
}

int main() {
    std::cout << math::add(3, 4) << '\n';
}
```

3.7.3 Using directive

```
using namespace math;
```

This allows direct access but should be used carefully, especially in header files.

3.7.4 Standard namespace

The standard library is contained in the `std` namespace:

```
#include <iostream>

int main() {
    std::cout << "Hello\n";
}
```

3.7.5 Best practice

Prefer explicit qualification:

```
std::cout << "Better practice\n";
```

Conclusion

A C++ program is a structured system built from functions, statements, scopes, declarations, and organized files. Understanding the role of `main`, the difference between declarations and definitions, the use of header and source files, and the purpose of namespaces provides the foundation for writing correct and scalable code.
This structural understanding is essential before moving into deeper topics such as types, memory, and object design.

Part II

Language Fundamentals

Variables and Basic Types

Variables and fundamental types form the core of every C++ program. Before a programmer can work with functions, classes, templates, containers, or algorithms, they must understand how values are represented, how objects are declared and initialized, how types affect behavior, and how to avoid unsafe conversions.

Modern C++ places great importance on correct initialization, explicit intent, and type safety. This is one of the reasons why learning variables in Modern C++ is not just about memorizing type names such as `int` or `double`. It is about understanding how values live in a program, how types guide the compiler, and how modern language features help the programmer write safer and clearer code.

This chapter introduces the most important language fundamentals related to variables and built-in types. It explains how to declare variables, how different initialization forms behave, how integer and floating-point types differ, how character and boolean types work, how `auto` performs type deduction, how `const`, `constexpr`, and `constinit` express different kinds of immutability and initialization constraints, and how type conversions should be handled carefully in professional code.

4.1 Declaring variables

A variable is a named object that stores a value of a specific type. In C++, every variable declaration introduces a name together with type information known to the compiler.

A simple declaration looks like this:

```
int count = 10;
double price = 19.95;
bool ready = true;
```

In these declarations:

- `int`, `double`, and `bool` are the types,
- `count`, `price`, and `ready` are the variable names,
- the expressions on the right initialize the variables.

A declaration gives the compiler enough information to understand:

- how much storage the object needs,
- what operations are valid on it,
- how the value should be interpreted.

4.1.1 Declaration before use

In C++, a name must be declared before it is used.

Incorrect:

```
int main() {
    value = 42;
}
```

This fails because `value` has never been declared.

Correct:

```
int main() {
    int value = 42;
}
```

4.1.2 Separate declaration and assignment

A variable may be declared first and assigned later:

```
int x;
x = 25;
```

However, in Modern C++, it is usually better to initialize a variable at the point of declaration whenever possible. This reduces the chance of using it in an uninitialized state.

Preferred:

```
int x = 25;
```

4.1.3 Multiple declarations in one statement

C++ allows multiple variables to be declared in one statement:

```
int a = 1, b = 2, c = 3;
```

This is legal, but many style guides prefer one declaration per line for readability, especially when types become more complex:

```
int a = 1;
int b = 2;
int c = 3;
```

4.1.4 Meaningful names

Variable names should communicate intent clearly.

Weak names:

```
int x = 7;
double y = 15.5;
```

Better names:

```
int retry_count = 7;
double item_price = 15.5;
```

Clear naming improves maintainability and reduces errors.

4.2 Modern initialization: `=`, `()` , `{}` and why brace initialization is preferred

C++ supports several initialization forms. Modern C++ programmers must understand the differences because initialization syntax affects safety, overload selection, and whether narrowing conversions are allowed.

4.2.1 Copy initialization with =

Example:

```
int a = 10;
double b = 3.14;
```

This syntax is familiar and widely used. Despite the name “copy initialization,” it does not necessarily imply a runtime copy for built-in types. It is simply a language form.

4.2.2 Direct initialization with parentheses

Example:

```
int a(10);
double b(3.14);
```

This form is also valid for built-in types and class types. However, in C++ code, parentheses can sometimes interact with parsing rules in ways that confuse beginners, especially with classes and constructors.

4.2.3 Brace initialization

Example:

```
int a{10};
double b{3.14};
```

Brace initialization, also called list initialization or uniform initialization in common teaching language, is one of the most important modern C++ forms.

4.2.4 Why brace initialization is preferred

Brace initialization is often preferred because:

- it provides a uniform syntax across many contexts,
- it helps avoid accidental narrowing conversions,
- it clearly expresses that initialization is happening,
- it works well with built-in types, aggregates, and many class types.

Example:

```
int a = 42;
int b(42);
int c{42};
```

All three are valid here. But braces provide extra safety in many situations.

4.2.5 Narrowing prevention

One of the strongest reasons to prefer brace initialization is that it rejects narrowing conversions that other forms may silently allow.
Allowed with =:

```
int x = 3.99;   // value becomes 3
```

Allowed with parentheses:

```
int y(3.99);    // value becomes 3
```

Rejected with braces:

```
int z{3.99};    // error: narrowing conversion
```

This behavior is one of the key safety improvements of modern initialization.

4.2.6 Empty brace initialization

Braces can also be used for value initialization:

```
int x{};
double y{};
bool flag{};
```

These become:

- `x == 0`
- `y == 0.0`
- `flag == false`

This is often safer than leaving built-in variables uninitialized.

4.2.7 Recommended style

For built-in types and most ordinary declarations, a strong modern default is:

```
int count{0};
double ratio{1.0};
bool enabled{true};
```

This makes initialization explicit and guards against accidental narrowing.

4.3 Integer and floating-point types

Fundamental arithmetic types are divided into integer types and floating-point types.

4.3.1 Integer types

Integer types represent whole numbers.
Common integer types include:

- `short`
- `int`
- `long`
- `long long`

Each of these also has unsigned forms:

- `unsigned short`
- `unsigned int`
- `unsigned long`
- `unsigned long long`

Example:

```
short s{12};
int i{1000};
long l{500000L};
long long big{9000000000LL};

unsigned int u{42};
```

4.3.2 Signed and unsigned

Signed integer types can represent negative and positive values. Unsigned integer types represent only non-negative values.
Example:

```
int temperature{-5};
unsigned int size{25};
```

Use unsigned types carefully. Although they may seem appropriate for values that cannot be negative, they can introduce surprising behavior in arithmetic and comparisons if mixed carelessly with signed types.

4.3.3 Choosing integer types

A practical beginner guideline is:

- use `int` for ordinary integer values,
- use `long long` for larger ranges when needed,
- use unsigned types only when their semantics are genuinely appropriate.

4.3.4 Floating-point types

Floating-point types represent numbers that may have fractional parts.
The three main floating-point types are:

- `float`
- `double`
- `long double`

Example:

```
float f{3.5f};
double d{3.141592653589793};
long double ld{2.718281828459045L};
```

4.3.5 Choosing floating-point types

A practical rule is:

- use `double` by default for ordinary floating-point calculations,
- use `float` when reduced precision or smaller storage is specifically desired,
- use `long double` only when you have a clear reason and understand implementation differences.

4.3.6 Integer versus floating-point behavior

Integer division discards any fractional part:

```
#include <iostream>

int main() {
    int a{7};
    int b{2};

    std::cout << a / b << '\n';   // prints 3
}
```

Floating-point division preserves fractional results:

```
#include <iostream>

int main() {
    double a{7.0};
    double b{2.0};

    std::cout << a / b << '\n';   // prints 3.5
}
```

This distinction is essential in real programs.

4.4 bool, char, wchar_t, char8_t, char16_t, char32_t

C++ provides special fundamental types for truth values and character representation.

4.4.1 bool

The bool type has two values:

- true
- false

Example:

```
bool ready{true};
bool failed{false};
```

Booleans are central to conditions and control flow:

```
#include <iostream>

int main() {
    bool logged_in{true};

    if (logged_in) {
        std::cout << "Access granted\n";
    }
}
```

4.4.2 char

The char type stores a character code unit. It is commonly used for narrow character data.
Example:

```
char letter{'A'};
char digit{'7'};
```

Characters are enclosed in single quotes.

4.4.3 wchar_t

The wchar_t type is a wide character type. Its size and exact representation are implementation-dependent. On Windows, it is commonly associated with UTF-16-based wide-character APIs.
Example:

```
wchar_t wc{L'Z'};
```

The prefix L indicates a wide character literal.

4.4.4 `char8_t`

`char8_t` was introduced in C++20 for UTF-8 code units as a distinct type.
Example:

```
char8_t c{u8'A'};
```

It is especially relevant when handling UTF-8 text in modern Unicode-aware code.

4.4.5 `char16_t`

`char16_t` is used for UTF-16 code units.
Example:

```
char16_t c{u'X'};
```

4.4.6 `char32_t`

`char32_t` is used for UTF-32 code units.
Example:

```
char32_t c{U'X'};
```

4.4.7 Character literals and prefixes

Common prefixes include:

- ordinary character literal: `'A'`
- wide character literal: `L'A'`
- UTF-8 character literal: `u8'A'`
- UTF-16 character literal: `u'A'`
- UTF-32 character literal: `U'A'`

4.4.8 A practical note

For ordinary English text in simple examples, `char` is common. For real Unicode-aware programs, text handling usually involves string types and encodings rather than isolated character variables. On Windows, it is important to be aware that platform APIs and library choices may interact with wide or UTF encodings differently.

4.5 Using `auto`

The `auto` keyword tells the compiler to deduce the variable type from the initializer.
Example:

```
auto x = 42;        // int
auto y = 3.14;      // double
auto flag = true;   // bool
```

4.5.1 Why `auto` is useful

`auto` is useful because:

- it avoids repeating long type names,
- it keeps declarations aligned with the actual initializer type,
- it avoids unintended conversions in many cases,
- it improves maintainability when types change.

Example with an iterator:

```
#include <vector>

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

    auto it = values.begin();
}
```

Without `auto`, the iterator type would be longer and less readable.

4.5.2 When `auto` is a good choice

`auto` is especially useful when:

- the initializer makes the type obvious,
- the exact type is verbose,
- you want the compiler to preserve the true type of an expression.

Example:

```
auto total = 10 + 20;      // deduced as int
auto average = 5.0 / 2.0;  // deduced as double
```

4.5.3 When explicit types may be better

Sometimes explicit types are clearer:

```
int retry_limit{5};
double interest_rate{0.05};
```

If the type itself is important information for the reader, writing it explicitly can be better than hiding it behind `auto`.

4.5.4 `auto` with braces

Care is needed with braces and `auto`, because deduction rules can differ depending on the form.
Example:

```
auto x = 42;     // int
auto y{42};      // int
```

For beginners, the safest habit is:

- use `auto name = expression;` for most ordinary `auto` declarations,
- use explicit types when teaching intent matters more than deduction convenience.

4.6 `const`, `constexpr`, `constinit`

These three keywords are related to immutability and initialization, but they are not the same.

4.6.1 `const`

`const` means the object cannot be modified through that name after initialization.
Example:

```
const int days_in_week{7};
```

Attempting to modify it causes an error:

```
const int value{10};
// value = 20;   // error
```

4.6.2 Important meaning of `const`

A `const` object is read-only through that variable, but `const` alone does not guarantee compile-time evaluation in every situation.
Example:

```
int get_value() {
    return 100;
}

int main() {
    const int x = get_value();   // valid, but may be run-time initialized
}
```

4.6.3 `constexpr`

`constexpr` is stronger. It requires initialization by a constant expression when applied to a variable, and such a variable is implicitly `const`.
Example:

```
constexpr int max_users{100};
constexpr double pi{3.141592653589793};
```

A `constexpr` variable must be initialized immediately with a compile-time constant expression.
Invalid example:

```
int get_value() {
    return 100;
}

// constexpr int x = get_value();   // error
```

4.6.4 `constexpr` functions

A function can also be declared `constexpr` if it is suitable for constant evaluation.

```
constexpr int square(int x) {
    return x * x;
}

int main() {
    constexpr int result = square(5);
}
```

Such functions may also execute at run time when called with non-constant arguments.

4.6.5 `constinit`

`constinit` was added in C++20. It applies to variables with static or thread storage duration and requires constant initialization, but it does not make the variable `const`.
Example:

```
constinit int global_counter = 0;

int main() {
    ++global_counter;
}
```

This is important because:

- the variable is guaranteed to be initialized at compile-time initialization phase,
- but it can still be modified later.

4.6.6 Difference between `constexpr` and `constinit`

`constexpr`:

- implies `const`,
- requires compile-time constant initialization,

- is used for values intended to remain constant.

`constinit`:

- does not imply `const`,
- requires constant initialization for static or thread-local objects,
- is used when early guaranteed initialization matters, but later mutation is still allowed.

Example comparison:

```
constexpr int fixed_value = 10;
// fixed_value = 20;   // error

constinit int global_value = 10;

int main() {
    global_value = 20;  // valid
}
```

4.6.7 Choosing between them

A practical rule is:

- use `const` for read-only values that do not need compile-time constant-expression rules,
- use `constexpr` for true compile-time constants,
- use `constinit` for static or thread-local objects that must be initialized in a guaranteed constant-initialization way but may still change later.

4.7 Type conversions

A type conversion changes a value from one type to another. Some conversions happen implicitly, while others are written explicitly.

4.7.1 Implicit conversions

C++ can convert between many arithmetic types automatically.
Example:

```
int i{42};
double d = i;   // implicit conversion from int to double
```

This conversion is usually safe because every `int` value in this example can be represented as a `double` within a wide range.

4.7.2 Potentially dangerous implicit conversions

Some conversions lose information:

```
double price{9.99};
int rounded = price;   // fractional part lost
```

Now `rounded` becomes `9`.

4.7.3 Explicit conversions

When a conversion is intentional, it should often be written explicitly.

Modern C++ strongly prefers `static_cast` for ordinary explicit numeric conversions:

```
double value{9.99};
int whole = static_cast<int>(value);
```

This makes intent visible to the reader and to code reviewers.

4.7.4 Why C-style casts should be avoided

C-style casts are harder to read and less specific:

```
int whole = (int)value;
```

Modern C++ prefers:

```
int whole = static_cast<int>(value);
```

4.7.5 Arithmetic promotions

In expressions, smaller integer types may be promoted automatically.

Example:

```
char a{10};
char b{20};

auto sum = a + b;   // usually promoted to int
```

This is one reason why expression types sometimes differ from the original variable types.

4.7.6 Mixed-type expressions

When integer and floating-point values are mixed, the integer is often converted to floating-point:

```
int count{3};
double factor{2.5};

double result = count * factor;   // count converted to double
```

Understanding these automatic conversions is important for correct reasoning about results.

4.8 Avoiding narrowing

A narrowing conversion is a conversion that may lose information, such as converting:

- from floating-point to integer,
- from a wider integer type to a narrower integer type,
- from a value whose range may not fit into the target type.

4.8.1 Examples of narrowing

```
double d{3.75};
int i = d;      // narrowing, fractional part lost

long long big{5000000000LL};
int small = static_cast<int>(big);   // possible loss of information
```

4.8.2 Why narrowing is dangerous

Narrowing conversions can silently:

- truncate values,
- change sign,
- overflow or wrap depending on the case,
- introduce subtle bugs that are difficult to detect.

4.8.3 Brace initialization as protection

Brace initialization helps prevent accidental narrowing:

```
int a{42};        // fine
// int b{3.14};   // error
```

This is one of the strongest practical reasons to favor braces in modern code.

4.8.4 If narrowing is truly intended

When a narrowing conversion is genuinely intended and understood, make it explicit:

```
double value{8.99};
int whole = static_cast<int>(value);
```

This does not make the conversion magically safe, but it makes the programmer's intent visible.

4.8.5 A safe design mindset

To avoid narrowing bugs:

- choose a type large enough for the required range,
- use brace initialization where practical,
- avoid silent arithmetic conversions when precision matters,
- use `static_cast` only when the conversion is deliberate and justified.

4.8.6 Practical examples

Unsafe style:

```
double tax_rate = 0.15;
int stored_rate = tax_rate;   // becomes 0
```

Safer design:

```
double tax_rate{0.15};
```

Another example:

```
int width{1920};
int height{1080};

double aspect_ratio = static_cast<double>(width) / height;
```

Here the explicit cast is helpful because it prevents integer division and clearly expresses the desired floating-point computation.

Conclusion

Variables and basic types are the first real building blocks of all C++ programs. A professional understanding of them goes far beyond memorizing a few type names. It includes knowing how to declare objects correctly, how to initialize them safely, how arithmetic types differ, how character and boolean types are represented, how `auto` helps the compiler deduce types, how `const`, `constexpr`, and `constinit` express different guarantees, and how to handle conversions carefully.

Modern C++ strongly encourages explicit intent and safer defaults. Brace initialization helps prevent narrowing. `auto` reduces unnecessary repetition. `constexpr` and `constinit` make initialization guarantees more precise. Explicit casts make dangerous conversions visible.

These ideas may seem simple at first, but they influence the correctness and quality of almost every program you will write. For that reason, mastering them early is one of the best investments a C++ learner can make.

Expressions and Operators

Expressions and operators are the core mechanisms through which C++ programs perform computation. Every meaningful piece of codewhether assigning values, evaluating conditions, or calling functions-relies on expressions.
An expression is a combination of values, variables, operators, and function calls that produces a result. Operators define how operands (values or variables) are combined and evaluated.
Modern C++ emphasizes writing expressions that are correct, clear, and free from unintended conversions or ambiguous behavior. This chapter explores arithmetic, logical, and comparison operators, operator precedence, increment and decrement semantics, implicit conversions, compound expressions, and best practices for writing clear and maintainable expressions.

5.1 Arithmetic operations

Arithmetic operators perform mathematical computations on numeric values.
The primary arithmetic operators are:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Remainder: `%`

5.1.1 Basic examples

```cpp
#include <iostream>

int main() {
    int a{10};
    int b{3};

    std::cout << a + b << '\n'; // 13
    std::cout << a - b << '\n'; // 7
    std::cout << a * b << '\n'; // 30
    std::cout << a / b << '\n'; // 3
    std::cout << a % b << '\n'; // 1
}
```

5.1.2 Integer vs floating-point division

Integer division discards the fractional part:

```
int x{7};
int y{2};

int result = x / y; // 3
```

Floating-point division preserves it:

```
double x{7.0};
double y{2.0};

double result = x / y; // 3.5
```

5.1.3 Remainder operator

The remainder operator `%` is defined for integer types:

```
int remainder = 10 % 4; // 2
```

5.1.4 Unary arithmetic operators

Unary operators act on a single operand:

```
int x{5};

int positive = +x;
int negative = -x;
```

5.2 Logical and comparison operators

Logical and comparison operators are essential for decision-making in programs.

5.2.1 Comparison operators

Comparison operators produce a `bool` result:

- Equal: `==`
- Not equal: `!=`
- Less than: `<`
- Greater than: `>`
- Less than or equal: `<=`
- Greater than or equal: `>=`

Example:

```
int a{10};
int b{20};

bool result = (a < b); // true
```

5.2.2 Logical operators

Logical operators combine boolean values:

- Logical AND: &&
- Logical OR: ||
- Logical NOT: !

Example:

```
bool is_ready{true};
bool is_valid{false};

bool result = is_ready && is_valid; // false
```

5.2.3 Short-circuit evaluation

Logical operators use short-circuit evaluation:

- &&: stops if the left side is false
- ||: stops if the left side is true

Example:

```
int x{0};

if (x != 0 && (10 / x) > 1) {
    // safe: second condition not evaluated
}
```

5.3 Operator precedence

Operator precedence determines the order in which parts of an expression are evaluated.

5.3.1 Basic precedence examples

```
int result = 2 + 3 * 4; // 14
```

Multiplication has higher precedence than addition.
To change evaluation order, use parentheses:

```
int result = (2 + 3) * 4; // 20
```

5.3.2 Common precedence hierarchy

From higher to lower (simplified):

- Unary operators (+, -, !, ++, −)
- Multiplicative (*, /, %)
- Additive (+, -)
- Relational (<, >, <=, >=)
- Equality (==, !=)
- Logical AND (&&)
- Logical OR (||)

5.3.3 Best practice

Even when precedence is known, parentheses should be used to improve clarity:

```
int result = (a + b) * c;
```

5.4 Increment / decrement

Increment and decrement operators modify a variable by 1.

- Increment: ++
- Decrement: −−

5.4.1 Prefix form

```
int x{5};
int y = ++x; // x = 6, y = 6
```

5.4.2 Postfix form

```
int x{5};
int y = x++; // y = 5, x = 6
```

5.4.3 Difference

- Prefix: increments first, then returns value
- Postfix: returns value, then increments

5.4.4 Best practice

Prefer prefix form when the result of the expression is not needed:

```
++x; // preferred
```

5.5 Implicit conversions

Implicit conversions happen automatically when types differ.

5.5.1 Example

```
int a{10};
double b{2.5};

double result = a + b; // a converted to double
```

5.5.2 Integer promotions

Smaller types are promoted:

```
char a{10};
char b{20};

int sum = a + b;
```

5.5.3 Boolean conversion

Non-zero values convert to true:

```
if (5) {
    // true
}
```

5.5.4 Risks

Implicit conversions can:

- lose precision
- change type unexpectedly
- introduce subtle bugs

5.6 Compound expressions

A compound expression combines multiple operations.
Example:

```
int result = (a + b) * (c - d);
```

5.6.1 Assignment expressions

Assignment is also an expression:

```
int x;
x = 5;
```

5.6.2 Compound assignment

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`

Example:

```
int x{10};
x += 5; // x = 15
```

5.6.3 Chained expressions

```
int a, b, c;
a = b = c = 10;
```

This works because assignment returns a value.

5.7 Writing clear expressions

Modern C++ encourages clarity over cleverness.

5.7.1 Avoid overly complex expressions

Bad:

```
int result = a + b * c - d / e + f * g;
```

Better:

```
int part1 = b * c;
int part2 = d / e;
int part3 = f * g;

int result = a + part1 - part2 + part3;
```

5.7.2 Use parentheses for clarity

```
int result = (a + b) * (c - d);
```

5.7.3 Avoid hidden side effects

Bad:

```
int result = x++ + ++y;
```

Better:

```
++y;
int result = x + y;
++x;
```

5.7.4 Prefer readability

Readable code is easier to maintain, debug, and review.

Conclusion

Expressions and operators define how computations are performed in C++. Understanding arithmetic, logical, and comparison operators, operator precedence, increment behavior, implicit conversions, and compound expressions is essential for writing correct programs.

Modern C++ encourages writing expressions that are clear, explicit, and safe. Avoid relying on subtle precedence rules or hidden conversions. Prefer readable and maintainable code, even if it requires more lines.

Mastering expressions is a fundamental step toward becoming a professional C++ developer.

Control Flow

Control flow determines the order in which statements execute in a C++ program. Without control flow, a program would simply execute one statement after another from top to bottom without making decisions, repeating tasks, or reacting to different conditions.
Modern C++ uses the same fundamental control-flow statements that have long existed in the language, but modern practice emphasizes writing them clearly, safely, and with strong attention to readability. A professional programmer must not only know how `if`, `switch`, and loops work, but must also know when each form is appropriate, how to avoid common mistakes, and how to express intent without unnecessary complexity.
This chapter introduces the main control-flow mechanisms used in everyday C++ programming:

- decision-making with `if`, `else if`, and `else`,
- multi-way branching with `switch`,
- repetition using `while`, `do-while`, and `for`,
- loop control with `break` and `continue`,
- readable iteration with the range-based `for` loop,
- practical guidance for choosing the right loop,
- common patterns and pitfalls.

6.1 `if`, `else if`, `else`

The `if` statement is the basic decision-making construct in C++. It executes one statement or block when a condition is true.

6.1.1 Basic `if`

```
#include <iostream>

int main() {
    int temperature{35};

    if (temperature > 30) {
        std::cout << "Hot weather\n";
    }
}
```

If the condition evaluates to `true`, the controlled statement or block executes. If the condition evaluates to `false`, execution continues after the `if` block.

6.1.2 `if` with `else`

The `else` branch provides an alternative path.

```
#include <iostream>

int main() {
    int age{16};

    if (age >= 18) {
        std::cout << "Adult\n";
    } else {
        std::cout << "Minor\n";
    }
}
```

Only one branch executes.

6.1.3 `else if`

When multiple conditions must be tested in sequence, `else if` is used.

```
#include <iostream>

int main() {
    int score{82};

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

The conditions are checked from top to bottom. As soon as one condition is true, its block executes and the rest are skipped.

6.1.4 Conditions are expressions

The condition inside `if` is an expression that is contextually converted to `bool`.

```
int value{5};

if (value) {
    // true because value is non-zero
}
```

Although this is valid, clearer code usually uses an explicit comparison when that improves intent.
Better:

```
if (value != 0) {
    // clearer intent
}
```

6.1.5 Braces and readability

C++ allows a single statement without braces:

```
if (ready)
    std::cout << "Ready\n";
```

However, modern style strongly prefers braces even for single-statement branches because:

- it improves readability,
- it reduces accidental mistakes during later edits,
- it keeps style consistent.

Preferred:

```
if (ready) {
    std::cout << "Ready\n";
}
```

6.1.6 Nested `if` statements

An `if` may appear inside another `if`:

```
#include <iostream>

int main() {
    bool logged_in{true};
    bool is_admin{false};

    if (logged_in) {
        if (is_admin) {
            std::cout << "Admin access\n";
        } else {
            std::cout << "Standard user access\n";
        }
    } else {
        std::cout << "Please log in\n";
    }
}
```

Nested logic is sometimes necessary, but deeply nested conditions often suggest that the code should be simplified or refactored.

6.1.7 A common pitfall: assignment instead of comparison

A frequent beginner error is writing = instead of ==.
Incorrect:

```
int x{10};

if (x = 5) {
    // wrong: assignment, not comparison
}
```

Correct:

```
if (x == 5) {
    // comparison
}
```

6.2 `switch`

The `switch` statement selects one branch among several possible branches based on the value of an integral or enumeration expression.

6.2.1 Basic `switch` structure

```
#include <iostream>

int main() {
    int day{3};

    switch (day) {
        case 1:
            std::cout << "Monday\n";
            break;
        case 2:
            std::cout << "Tuesday\n";
            break;
        case 3:
            std::cout << "Wednesday\n";
            break;
        default:
            std::cout << "Unknown day\n";
            break;
    }
}
```

The controlling expression is evaluated once. Execution jumps to the matching `case` label.

6.2.2 Why `break` is usually required

Without `break`, execution continues into the next case. This is called fallthrough.
Example without `break`:

```cpp
#include <iostream>

int main() {
    int value{1};

    switch (value) {
        case 1:
            std::cout << "One\n";
        case 2:
            std::cout << "Two\n";
        default:
            std::cout << "Done\n";
    }
}
```

Output:

```
One
Two
Done
```

This is often a bug.

6.2.3 Intentional fallthrough

Sometimes fallthrough is intentional, especially when multiple cases share logic.

```cpp
#include <iostream>

int main() {
    char command{'y'};

    switch (command) {
        case 'y':
        case 'Y':
            std::cout << "Confirmed\n";
            break;
        case 'n':
        case 'N':
            std::cout << "Rejected\n";
            break;
        default:
            std::cout << "Unknown response\n";
            break;
    }
}
```

Here the grouped cases are clear and safe.

6.2.4 `default`

The `default` label handles values not matched by any explicit case.

```
default:
    std::cout << "Invalid option\n";
    break;
```

Using `default` is generally a good idea unless there is a specific reason not to.

6.2.5 When `switch` is a good choice

`switch` is often appropriate when:

- a single variable is compared against many constant values,
- the set of options is discrete and known,
- the alternatives are simpler to read as case labels than as many `else if` comparisons.

6.2.6 When `if` may be better

`if` is usually better when:

- conditions are based on ranges,
- comparisons involve different variables,
- logic is not simple equality against constant values,
- the conditions are more expressive than a case list.

6.3 `while`, `do-while`, `for`

Iteration statements repeat code while a condition holds or for a defined sequence of steps.

6.3.1 `while`

A `while` loop checks its condition before each iteration. If the condition is false from the beginning, the loop body may execute zero times.

```
#include <iostream>

int main() {
    int count{1};

    while (count <= 5) {
        std::cout << count << '\n';
        ++count;
    }
}
```

This is useful when the number of iterations is not best expressed as a fixed counter loop.

6.3.2 `do-while`

A `do-while` loop executes the body first, then checks the condition. Therefore, the body executes at least once.

```
#include <iostream>

int main() {
    int value{1};

    do {
        std::cout << value << '\n';
        ++value;
    } while (value <= 3);
}
```

This form is useful when at least one execution is required before testing continuation.

6.3.3 `for`

A traditional `for` loop combines initialization, condition, and iteration step in one structure.

```
#include <iostream>

int main() {
    for (int i{0}; i < 5; ++i) {
        std::cout << i << '\n';
    }
}
```

This is often the clearest loop when counting or stepping through a numeric range.

6.3.4 Understanding the `for` structure

A traditional `for` loop has the form:

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

The order of behavior is:

1. perform initialization once,
2. test the condition,
3. execute the body if true,
4. perform the update,
5. repeat.

6.3.5 Equivalent `while` view

A `for` loop can often be understood as a structured `while` loop.

```
for (int i{0}; i < 3; ++i) {
    std::cout << i << '\n';
}
```

Roughly corresponds to:

```
int i{0};
while (i < 3) {
    std::cout << i << '\n';
    ++i;
}
```

6.4 `break`, `continue`

The `break` and `continue` statements modify normal loop or `switch` flow.

6.4.1 `break`

`break` exits the nearest enclosing loop or `switch` statement immediately.
Example in a loop:

```
#include <iostream>

int main() {
    for (int i{0}; i < 10; ++i) {
        if (i == 5) {
            break;
        }

        std::cout << i << '\n';
    }
}
```

This prints:

```
0
1
2
3
4
```

6.4.2 `continue`

`continue` skips the rest of the current loop iteration and proceeds to the next iteration.

```
#include <iostream>

int main() {
```

```
    for (int i{0}; i < 6; ++i) {
        if (i == 3) {
            continue;
        }

        std::cout << i << '\n';
    }
}
```

This prints:

```
0
1
2
4
5
```

6.4.3 Use with care

Both `break` and `continue` are useful, but overusing them can make loop logic harder to follow. Prefer clear loop conditions when practical.

6.4.4 Example: searching in a loop

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

int main() {
    std::vector<int> values{4, 8, 15, 16, 23, 42};
    int target{16};
    bool found{false};

    for (int value : values) {
        if (value == target) {
            found = true;
            break;
        }
    }

    if (found) {
        std::cout << "Target found\n";
    } else {
        std::cout << "Target not found\n";
    }
}
```

This is a clear and appropriate use of `break`.

6.5 Range-based `for` loop

The range-based `for` loop, introduced in C++11, is one of the most useful and readable loop forms in Modern C++. It iterates directly over the elements of a range such as an array, container, or other range-providing object.

6.5.1 Basic example

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

int main() {
    std::vector<int> values{10, 20, 30, 40};

    for (int value : values) {
        std::cout << value << '\n';
    }
}
```

This is usually clearer than writing an index-based loop when the index itself is not needed.

6.5.2 Read-only reference form

When elements should not be copied unnecessarily, use a const reference:

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

int main() {
    std::vector<std::string> names{"Ayman", "Omar", "Sara"};

    for (const std::string& name : names) {
        std::cout << name << '\n';
    }
}
```

This avoids copying each string.

6.5.3 Modifying elements

To modify the actual elements in the range, use a non-const reference:

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

int main() {
    std::vector<int> values{1, 2, 3, 4};

    for (int& value : values) {
        value *= 10;
    }

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

6.5.4 Arrays and strings

Range-based `for` works with arrays as well:

```
#include <iostream>

int main() {
    int data[]{5, 10, 15, 20};

    for (int value : data) {
        std::cout << value << '\n';
    }
}
```

6.5.5 When range-based `for` is best

This loop is often the best choice when:

- every element must be visited,
- the loop does not need an index,
- the goal is simple iteration over a range,
- readability matters more than manual control details.

6.6 Choosing the right loop

Professional code becomes easier to read when the chosen loop form matches the intent of the task.

6.6.1 Choose `for` for counted iteration

When the loop is naturally about a counter, the traditional `for` loop is often the clearest.

```
for (int i{0}; i < 10; ++i) {
    std::cout << i << '\n';
}
```

6.6.2 Choose `while` for condition-driven repetition

When repetition depends on a condition whose end point is not best expressed as a fixed counter, `while` often reads more naturally.

```
int attempts{0};

while (attempts < 3) {
    ++attempts;
}
```

6.6.3 Choose `do-while` when at least one execution is required

This is common for menu loops or input validation patterns.

```
#include <iostream>

int main() {
    int choice{};

    do {
        std::cout << "Enter 1 to continue, 0 to exit: ";
        std::cin >> choice;
    } while (choice != 0 && choice != 1);
}
```

6.6.4 Choose range-based `for` for container traversal

When iterating through all elements of a container, range-based `for` is usually the first choice.

```
for (const auto& item : items) {
    // process item
}
```

6.6.5 Do not force one loop everywhere

A strong programmer does not use the same loop for every task. The goal is not personal habit. The goal is matching structure to meaning.

6.7 Common patterns and pitfalls

Control flow is easy to begin using, but many common bugs appear here. Learning them early prevents many later problems.

6.7.1 Pattern: counting loop

```
for (int i{1}; i <= 10; ++i) {
    std::cout << i << '\n';
}
```

This is a standard counting pattern.

6.7.2 Pattern: processing a container

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

int main() {
    std::vector<int> values{2, 4, 6, 8};

    for (int value : values) {
        std::cout << value * 2 << '\n';
    }
}
```

6.7.3 Pattern: sentinel-controlled loop

A sentinel value ends the loop.

```
#include <iostream>

int main() {
    int value{};

    while (true) {
        std::cin >> value;

        if (value == -1) {
            break;
        }

        std::cout << "You entered: " << value << '\n';
    }
}
```

6.7.4 Pitfall: off-by-one errors

These are among the most common loop bugs.

Incorrect:

```
for (int i{0}; i <= 5; ++i) {
    // executes 6 times
}
```

If the intention was five iterations, the condition should be:

```
for (int i{0}; i < 5; ++i) {
    // executes 5 times
}
```

6.7.5 Pitfall: infinite loops

A loop that never updates its condition variable may never end.

Incorrect:

```
int count{0};

while (count < 5) {
    std::cout << count << '\n';
}
```

Correct:

```
int count{0};

while (count < 5) {
    std::cout << count << '\n';
    ++count;
}
```

6.7.6 Pitfall: modifying copies instead of elements

In a range-based loop, this does not modify the original container:

```
for (int value : values) {
    value *= 2;
}
```

Use a reference to modify actual elements:

```
for (int& value : values) {
    value *= 2;
}
```

6.7.7 Pitfall: missing `break` in `switch`

This is one of the classic mistakes in branching logic.

Incorrect:

```
switch (option) {
    case 1:
        std::cout << "Option 1\n";
    case 2:
        std::cout << "Option 2\n";
}
```

Correct:

```
switch (option) {
    case 1:
        std::cout << "Option 1\n";
        break;
    case 2:
        std::cout << "Option 2\n";
        break;
}
```

6.7.8 Pitfall: deeply nested control flow

Too much nesting reduces readability.

Harder to read:

```
if (logged_in) {
    if (account_active) {
        if (has_permission) {
            std::cout << "Access granted\n";
        }
    }
}
```

Often this can be simplified:

```
if (!logged_in) {
    return 0;
}

if (!account_active) {
    return 0;
}

if (!has_permission) {
    return 0;
}

std::cout << "Access granted\n";
```

6.7.9 Pitfall: unclear loop intent

A counted loop used only for element traversal is often less clear than a range-based loop.
Less clear:

```
for (std::size_t i{0}; i < values.size(); ++i) {
    std::cout << values[i] << '\n';
}
```

Clearer when the index is unnecessary:

```
for (int value : values) {
    std::cout << value << '\n';
}
```

Conclusion

Control flow is one of the essential foundations of programming in C++. The `if` family allows programs to make decisions. `switch` provides structured multi-way branching. `while`, `do-while`, and `for` enable repetition. `break` and `continue` refine loop control. The range-based `for` loop makes iteration over containers and arrays more readable and idiomatic in Modern C++.
Professional use of control flow is not only about knowing syntax. It is about choosing the right construct for the task, writing conditions clearly, preventing fallthrough mistakes, avoiding infinite loops, and keeping logic readable as programs grow.
A reader who masters these control-flow tools well will find that almost every later topic in C++ becomes easier to express and easier to reason about.

Modern Input and Output

Input and output are among the first practical capabilities every programmer uses. A program that cannot display results or receive data from the user remains isolated. In C++, input and output have evolved through multiple styles and libraries. The classic stream-based model remains important and widely used, while the modern formatting model introduced by newer standards offers a cleaner and more expressive way to build formatted text.

A beginner should learn both approaches. The stream-based approach is fundamental because it appears throughout existing codebases, standard library usage, and older teaching materials. The modern formatting approach is equally important because it often produces clearer and more maintainable formatting code.

This chapter introduces the most important facilities for text input and output in Modern C++:

- standard stream objects such as `std::cout`, `std::cin`, and `std::cerr`,
- modern formatting with `std::format`,
- modern output with `std::print`,
- working with strings in output,
- formatting numbers and aligned text,
- choosing between iostream-style formatting and the newer formatting model.

All examples are suitable for Windows development. For ordinary stream examples, include `<iostream>`. For `std::format`, include `<format>` and compile in a C++20-capable mode. For `std::print`, include `<print>` and use a toolchain mode and standard library implementation that provide the C++23 printing facilities.

7.1 `std::cout`, `std::cin`, `std::cerr`

The header `<iostream>` provides the standard stream objects used for console-style input and output.

The three most important objects for beginners are:

- `std::cout` for normal output,
- `std::cin` for input,
- `std::cerr` for error output.

7.1.1 `std::cout`

`std::cout` is the standard output stream. It is commonly used to display normal program output to the console.

```
#include <iostream>

int main() {
    std::cout << "Welcome to Modern C++\n";
    std::cout << "This is standard output\n";
}
```

The insertion operator « sends values into the output stream.

```
#include <iostream>

int main() {
    int age{25};
    double price{19.95};

    std::cout << "Age: " << age << '\n';
    std::cout << "Price: " << price << '\n';
}
```

7.1.2 `std::cin`

`std::cin` is the standard input stream. It is commonly used to read values from the keyboard.

```
#include <iostream>

int main() {
    int number{};

    std::cout << "Enter a number: ";
    std::cin >> number;

    std::cout << "You entered: " << number << '\n';
}
```

The extraction operator » reads formatted input from the stream into a variable.

7.1.3 Basic input of multiple values

```
#include <iostream>

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

    std::cout << "Enter two integers: ";
    std::cin >> a >> b;

    std::cout << "Sum = " << (a + b) << '\n';
}
```

7.1.4 Input failure

Input may fail if the user types a value that does not match the expected type.

```
#include <iostream>

int main() {
    int value{};

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

    if (!std::cin) {
        std::cerr << "Input error: expected an integer\n";
        return 1;
    }

    std::cout << "Valid input: " << value << '\n';
}
```

This is an important habit: check input when correctness matters.

7.1.5 `std::cerr`

`std::cerr` is the standard error stream. It is intended for error messages and diagnostics.

```
#include <iostream>

int main() {
    std::cerr << "Error: file could not be opened\n";
}
```

A practical pattern is to send normal results to `std::cout` and problems to `std::cerr`.

```
#include <iostream>

int main() {
    bool file_opened{false};

    if (!file_opened) {
        std::cerr << "Error: unable to open configuration file\n";
        return 1;
    }

    std::cout << "Configuration loaded successfully\n";
}
```

7.1.6 Newline versus `std::endl`

A newline can be written using `'\n'`:

```
std::cout << "Hello\n";
```

Or using `std::endl`:

```
std::cout << "Hello" << std::endl;
```

The difference is important:

- `'\n'` writes a newline character,
- `std::endl` writes a newline and flushes the stream.

For most ordinary output, `'\n'` is preferable because unnecessary flushing may reduce performance.

7.1.7 Flushing output

Sometimes immediate output is necessary, especially in interactive console programs.

```
#include <iostream>

int main() {
    std::cout << "Processing..." << std::flush;
}
```

This forces the stream to flush without inserting a newline.

7.1.8 A small interactive example

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

int main() {
    std::string name;
    int age{};

    std::cout << "Enter your name: ";
    std::cin >> name;

    std::cout << "Enter your age: ";
    std::cin >> age;

    if (!std::cin) {
        std::cerr << "Invalid age input\n";
        return 1;
    }

    std::cout << "Hello, " << name << ". You are " << age << " years old.\n";
}
```

7.2 `std::format` and `std::print`

Modern C++ introduced a new formatting model designed to make formatted text clearer and more expressive than stateful stream formatting.

7.2.1 `std::format`

`std::format` creates a formatted string using replacement fields inside a format string.

Basic example:

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

int main() {
    std::string name{"Ayman"};
    int score{95};

    std::string message = std::format("Student: {}, Score: {}", name, score);
    std::cout << message << '\n';
}
```

This approach is easier to read than long chains of stream insertion when formatting becomes more complex.

7.2.2 Why `std::format` is useful

`std::format` is useful because:

- the format string shows the final text structure clearly,
- formatting options are localized in one place,
- it avoids persistent stream formatting state,
- it often reads more naturally for mixed text and values.

7.2.3 Formatting multiple types

```
#include <format>
#include <iostream>

int main() {
    int id{1001};
    double price{45.75};
    bool available{true};

    std::cout << std::format(
        "ID: {}, Price: {}, Available: {}",
        id, price, available
    ) << '\n';
}
```

7.2.4 Positional formatting

Arguments may also be referenced by index.

```cpp
#include <format>
#include <iostream>

int main() {
    std::cout << std::format("{1} scored {0} points", 98, "Omar") << '\n';
}
```

7.2.5 `std::print`

`std::print` writes formatted output directly without requiring a separate `std::cout << std::format(...)` step.
Basic example:

```cpp
#include <print>

int main() {
    std::print("Welcome to {}\n", "Modern C++");
}
```

This is one of the cleanest output styles in newer C++.

7.2.6 `std::println`

When available in the implementation, `std::println` is a convenient variant that appends a newline.

```cpp
#include <print>

int main() {
    std::println("User {} logged in", "Ayman");
}
```

7.2.7 When to use `std::format` versus `std::print`

A practical rule is:

- use `std::format` when you want to build a string first,
- use `std::print` when you want to write formatted text directly to output.

Example of building a string first:

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

int main() {
    std::string log_line = std::format("User {} connected from {}", "Ayman", "127.0.0.1");

    std::ofstream file("app.log");
    file << log_line << '\n';
}
```

Example of direct output:

```
#include <print>

int main() {
    std::print("Temperature: {} C\n", 23.5);
}
```

7.2.8 Windows compilation note

For Windows development:

- compile `std::format` examples in a C++20-capable mode,
- compile `std::print` examples in a toolchain mode and standard library implementation that provide the C++23 printing facilities.

Example command-line styles:

```
cl /EHsc /std:c++20 format_example.cpp
cl /EHsc /std:c++latest print_example.cpp
```

7.3 Working with strings in output

Strings are central to almost all real-world output. Modern C++ commonly uses `std::string` for ordinary text and combines it naturally with both streams and the modern formatting facilities.

7.3.1 Outputting `std::string` with streams

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

int main() {
    std::string title{"Modern C++ From Zero to Professional"};
    std::cout << "Book title: " << title << '\n';
}
```

7.3.2 Concatenating strings for output

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

int main() {
    std::string first{"Modern"};
    std::string second{"C++"};

    std::string combined = first + " " + second;
    std::cout << combined << '\n';
}
```

7.3.3 Using `std::format` with strings

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

int main() {
    std::string user{"Ayman"};
    std::string role{"Author"};

    std::cout << std::format("User: {}, Role: {}", user, role) << '\n';
}
```

7.3.4 Quoted strings with iostream manipulators

Sometimes it is useful to print strings with quotes around them, especially for logs or debugging.

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

int main() {
    std::string file_name{"report.txt"};
    std::cout << std::quoted(file_name) << '\n';
}
```

7.3.5 Reading a full line with `std::getline`

`std::cin » name;` stops at whitespace. To read a full line, use `std::getline`.

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

int main() {
    std::string full_name;

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

    std::cout << "Hello, " << full_name << '\n';
}
```

7.3.6 A common pitfall with `std::getline`

If `std::getline` follows formatted extraction with », a leftover newline may need to be handled.
Problematic version:

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

int main() {
    int age{};
```

```
    std::string full_name;

    std::cin >> age;
    std::getline(std::cin, full_name); // may read an empty line
}
```

Safer version:

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

int main() {
    int age{};
    std::string full_name;

    std::cout << "Enter age: ";
    std::cin >> age;

    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

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

    std::cout << "Name: " << full_name << ", Age: " << age << '\n';
}
```

7.4 Formatting text and numbers

Formatted output is one of the areas where the contrast between iostreams and the modern formatting model becomes very visible.

7.4.1 Formatting with iostream manipulators

The header `<iomanip>` provides common manipulators such as:

- `std::setw`
- `std::setfill`
- `std::setprecision`
- `std::fixed`
- `std::scientific`
- `std::hex`
- `std::dec`
- `std::left`
- `std::right`

7.4.2 Width and alignment

```
#include <iomanip>
#include <iostream>

int main() {
    std::cout << std::setw(10) << 42 << '\n';
    std::cout << std::left << std::setw(10) << 42 << "<\n";
    std::cout << std::right << std::setw(10) << 42 << "<\n";
}
```

7.4.3 Fill characters

```
#include <iomanip>
#include <iostream>

int main() {
    std::cout << std::setfill('0') << std::setw(8) << 42 << '\n';
}
```

7.4.4 Floating-point precision

```
#include <iomanip>
#include <iostream>

int main() {
    double value{123.456789};

    std::cout << std::setprecision(4) << value << '\n';
    std::cout << std::fixed << std::setprecision(2) << value << '\n';
    std::cout << std::scientific << std::setprecision(3) << value << '\n';
}
```

7.4.5 Integer radix formatting

```
#include <iostream>
#include <iomanip>

int main() {
    int value{255};

    std::cout << "Decimal: " << std::dec << value << '\n';
    std::cout << "Hex: " << std::hex << value << '\n';
    std::cout << "Octal: " << std::oct << value << '\n';
}
```

7.4.6 Important note about stream state

A major characteristic of iostream formatting is that formatting state often persists until changed.

```
#include <iomanip>
#include <iostream>
```

```
int main() {
    double value{12.34567};

    std::cout << std::fixed << std::setprecision(2);
    std::cout << value << '\n';
    std::cout << value << '\n'; // still fixed with precision 2
}
```

This persistence is powerful, but it can also make code harder to reason about if formatting changes are scattered through a large function.

7.4.7 Formatting with `std::format`

Modern formatting expresses width, alignment, fill, precision, and representation directly inside the format string.

```
#include <format>
#include <iostream>

int main() {
    int value{42};
    double pi{3.1415926535};

    std::cout << std::format("Value right aligned: {:>8}\n", value);
    std::cout << std::format("Value zero filled: {:0>8}\n", value);
    std::cout << std::format("Pi with 2 decimals: {:.2f}\n", pi);
    std::cout << std::format("Pi scientific: {:.3e}\n", pi);
}
```

7.4.8 Readable column formatting

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

struct item {
    std::string name;
    double price;
};

int main() {
    std::vector<item> items{
        {"Keyboard", 120.5},
        {"Mouse", 75.0},
        {"Monitor", 999.99}
    };

    std::cout << std::format("{:<15} {:>10}\n", "Item", "Price");
    std::cout << std::format("{:-<15} {:-<10}\n", "", "");

    for (const auto& x : items) {
```

```
        std::cout << std::format("{:<15} {:>10.2f}\n", x.name, x.price);
    }
}
```

7.4.9 Combining text and numbers cleanly

```
#include <format>
#include <iostream>

int main() {
    std::string city{"Dammam"};
    int year{2026};
    double temperature{31.75};

    std::cout << std::format(
        "City: {}, Year: {}, Temperature: {:.1f} C\n",
        city, year, temperature
    );
}
```

7.5 iostream vs modern formatting approach

Both approaches are important, and both are worth learning. But they serve different styles of expression.

7.5.1 iostream approach

Classic stream output:

```
#include <iostream>
#include <iomanip>

int main() {
    std::string name{"Ayman"};
    int score{97};
    double average{91.375};

    std::cout << "Name: " << name
              << ", Score: " << score
              << ", Average: " << std::fixed << std::setprecision(2)
              << average << '\n';
}
```

Advantages:

- fundamental and widely supported,
- natural for incremental insertion,
- works well with existing stream-based code,
- easy for simple output chains.

Disadvantages:

- formatting state persists,
- complex formatting becomes verbose,
- alignment and precision logic may spread across multiple operators and manipulators,
- final output shape is sometimes harder to see at a glance.

7.5.2 Modern formatting approach

Equivalent style using `std::format`:

```
#include <format>
#include <iostream>

int main() {
    std::string name{"Ayman"};
    int score{97};
    double average{91.375};

    std::cout << std::format(
        "Name: {}, Score: {}, Average: {:.2f}\n",
        name, score, average
    );
}
```

Or with direct printing:

```
#include <print>

int main() {
    std::println("Name: {}, Score: {}, Average: {:.2f}",
                 "Ayman", 97, 91.375);
}
```

Advantages:

- output structure is visible directly in the format string,
- width, precision, and alignment are localized and easy to read,
- no persistent stream formatting state,
- often more convenient for structured human-readable output.

Disadvantages:

- requires newer standard-library support,
- existing legacy codebases may still rely heavily on streams,
- stream-oriented patterns still remain necessary in many contexts.

7.5.3 When to choose each one

A practical modern guideline is:

- use `std::cout` and classic streams for simple incremental output, stream-oriented APIs, or when working in environments where streams are already the dominant style,
- use `std::format` or `std::print` when formatting becomes structured, alignment matters, or readability of the final text layout is important.

7.5.4 A side-by-side example

iostream style:

```
#include <iomanip>
#include <iostream>

int main() {
    int id{12};
    double amount{345.678};

    std::cout << "ID: " << std::setw(6) << id
              << " Amount: " << std::fixed << std::setprecision(2)
              << amount << '\n';
}
```

Modern formatting style:

```
#include <format>
#include <iostream>

int main() {
    int id{12};
    double amount{345.678};

    std::cout << std::format("ID: {:>6} Amount: {:.2f}\n", id, amount);
}
```

The second version often makes the layout easier to read immediately.

7.5.5 A practical recommendation for this book

For a beginner learning Modern C++ correctly:

- first become comfortable with `std::cout`, `std::cin`, and `std::cerr`,
- then learn `std::format` as the modern formatting tool for building strings and formatted output,
- adopt `std::print` when your toolchain provides it and when direct formatted output improves clarity.

This progression keeps the learner grounded in the traditional standard stream model while moving naturally toward the modern formatting style.

Conclusion

Modern C++ input and output can be understood as two closely related layers.
The first layer is the classic stream-based model built around `std::cout`, `std::cin`, and `std::cerr`. This model remains foundational, especially for console interaction, stream-based libraries, and existing codebases. It is flexible and powerful, but complex formatting can become verbose because formatting state is persistent and spread across manipulators.
The second layer is the newer formatting model built around `std::format` and `std::print`. This approach often makes formatted output clearer because the text structure, alignment, width, and precision appear directly in the format string. For many output tasks, especially those involving structured textual presentation, it is the cleaner modern approach.
A strong C++ programmer should know both. Streams remain essential. Modern formatting improves clarity. Learning when to use each is part of writing professional C++ code.

Part III

Functions and Program Structure

Functions

Functions are one of the most important structural tools in C++. They allow a program to be divided into smaller, reusable, and meaningful units. A well-designed function performs one clear task, exposes a clean interface, and can be called from different parts of a program without duplicating logic.

Without functions, even medium-sized programs quickly become difficult to read, test, maintain, and extend. With functions, a programmer can organize work into logical operations such as reading input, validating data, computing a result, printing output, or transforming objects.

Modern C++ continues to rely heavily on functions, but it also places stronger emphasis on writing function interfaces carefully. The choice between pass by value, pass by reference, and pass by `const&` affects both correctness and performance. Features such as overloading, default arguments, inline functions, and forward declarations make programs more expressive and modular when used properly.

This chapter introduces the core concepts that every serious C++ programmer must understand about functions:

- how functions are defined and called,
- how parameters and return values work,
- the meaning of pass by value and pass by reference,
- why `const&` is so important in modern code,
- how default arguments and overloads are used,
- what `inline` really means,
- why forward declarations are necessary in multi-file programs.

8.1 Function definition and calls

A function definition provides the complete implementation of a function. It specifies:

- the return type,
- the function name,
- the parameter list,
- the function body.

A simple function definition looks like this:

```cpp
#include <iostream>

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

int main() {
    int result = add(10, 20);
    std::cout << result << '\n';
}
```

In this example:

- `int` before `add` is the return type,
- `add` is the function name,
- `int a, int b` are the parameters,
- the braces contain the body,
- `return a + b;` sends a value back to the caller.

A function call invokes the function by name and provides arguments:

```cpp
int result = add(10, 20);
```

Here:

- `10` is passed to parameter `a`,
- `20` is passed to parameter `b`,
- the return value is stored in `result`.

8.1.1 Execution flow of a function call

When a function call occurs:

1. control transfers from the caller to the called function,
2. the arguments are associated with the parameters,
3. the function body executes,
4. a return statement may produce a result,
5. control returns to the caller.

8.1.2 Functions with no return value

Some functions perform work without returning a value. Such functions use the return type `void`.

```
#include <iostream>

void print_banner() {
    std::cout << "Modern C++ Book\n";
}

int main() {
    print_banner();
}
```

A `void` function may still use `return;` to exit early, but it does not return a value.

```
#include <iostream>

void validate(int value) {
    if (value < 0) {
        std::cout << "Negative value is not allowed\n";
        return;
    }

    std::cout << "Accepted value: " << value << '\n';
}
```

8.1.3 A function should have a clear purpose

A beginner often writes overly large functions. A better design is to keep each function focused.
Less structured style:

```
#include <iostream>

int main() {
    int a{10};
    int b{20};

    int sum = a + b;
    std::cout << "Sum = " << sum << '\n';

    int diff = a - b;
    std::cout << "Difference = " << diff << '\n';
}
```

More structured style:

```
#include <iostream>

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

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

int main() {
    std::cout << "Sum = " << add(10, 20) << '\n';
    std::cout << "Difference = " << subtract(10, 20) << '\n';
}
```

This makes the program easier to read and reuse.

8.2 Parameters and return values

Parameters define the inputs a function accepts. Return values define the output a function produces.

8.2.1 Parameters

A parameter is a named object introduced in the function's parameter list.

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

Here `x` and `y` are parameters.

8.2.2 Arguments versus parameters

The distinction is important:

- parameters appear in the function definition or declaration,
- arguments are the actual values supplied at the call site.

```
double result = multiply(2.5, 4.0);
```

In this call:

- `x` and `y` are parameters,
- `2.5` and `4.0` are arguments.

8.2.3 Return values

A function may return a value using `return`.

```
int square(int x) {
    return x * x;
}
```

The expression after `return` must be compatible with the declared return type.

8.2.4 Returning different values from different branches

```
int maximum(int a, int b) {
    if (a > b) {
        return a;
    }

    return b;
}
```

8.2.5 Returning objects

Functions can return objects, not just built-in values.

```
#include <string>

std::string make_title() {
    return "Modern C++";
}
```

8.2.6 Why return values are powerful

Returning values helps functions remain composable.

```
#include <iostream>

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

int square(int x) {
    return x * x;
}

int main() {
    int result = square(add(2, 3));
    std::cout << result << '\n';
}
```

Here the result of one function becomes the input of another.

8.3 Pass by value

Pass by value means that the function receives its own copy of the argument.

```
#include <iostream>

void increment(int x) {
    ++x;
    std::cout << "Inside function: " << x << '\n';
}
```

```
int main() {
    int value{10};
    increment(value);
    std::cout << "In main: " << value << '\n';
}
```

Output conceptually:

```
Inside function: 11
In main: 10
```

The original variable in `main` remains unchanged because the function modifies only its local copy.

8.3.1 When pass by value is appropriate

Pass by value is usually appropriate when:

- the argument type is small and inexpensive to copy,
- the function should not modify the caller's object,
- the function needs its own independent copy.

Examples of small types:

- `int`
- `double`
- `char`
- `bool`
- many small enums

8.3.2 A good pass-by-value example

```
int cube(int x) {
    return x * x * x;
}
```

Passing an `int` by value is simple and efficient.

8.3.3 Pass by value with larger objects

Pass by value also works for class types, but it copies the object.

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

void print_name(std::string name) {
    std::cout << name << '\n';
}
```

This is valid, but the string is copied. For larger objects, that may be unnecessary.

8.3.4 A practical modern rule

Use pass by value for small, cheap-to-copy types unless you have a strong reason not to.

8.4 Pass by reference

Pass by reference allows a function parameter to refer directly to the caller's object instead of receiving a copy.

```
#include <iostream>

void increment(int& x) {
    ++x;
}

int main() {
    int value{10};
    increment(value);
    std::cout << value << '\n';
}
```

Now the original `value` becomes `11` because the parameter `x` refers to the same object.

8.4.1 What a reference means

A reference behaves like an alias to an existing object. It is not a separate copy.

```
int a{5};
int& ref = a;

ref = 20;
// a is now 20
```

8.4.2 When pass by reference is useful

Pass by reference is useful when:

- the function must modify the caller's object,
- copying would be unnecessary or expensive,
- the parameter semantically represents something to be updated.

8.4.3 Modifying multiple values

```
#include <iostream>

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

```
int main() {
    int x{10};
    int y{20};

    swap_values(x, y);

    std::cout << x << ' ' << y << '\n';
}
```

8.4.4 Reference parameters should express intent clearly

A non-`const` reference signals that the function may modify the argument. This is important interface information. A caller should be able to see from the signature that modification is possible.

8.5 `const&`

One of the most important parameter forms in Modern C++ is the const reference, written as `const T&`.

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

void print_name(const std::string& name) {
    std::cout << name << '\n';
}
```

This means:

- no copy of the argument is made,
- the function cannot modify the argument through that parameter,
- the call remains efficient for large objects.

8.5.1 Why `const&` is so common

For many class types, copying may be more expensive than passing a reference. At the same time, many functions do not need to modify the argument. In such cases, `const&` is often the best choice.

```
#include <string>

bool is_empty(const std::string& text) {
    return text.empty();
}
```

8.5.2 Example with a vector

```
#include <vector>

int sum(const std::vector<int>& values) {
    int total{0};
```

```
    for (int value : values) {
        total += value;
    }

    return total;
}
```

Passing the vector by value would copy the entire vector. Passing by `const&` avoids that.

8.5.3 What `const&` prevents

This code would fail:

```
#include <string>

void modify(const std::string& text) {
    // text += "x"; // error
}
```

That is exactly the point: the interface promises read-only access.

8.5.4 A practical guideline

A good rule of thumb is:

- pass small scalar types by value,
- pass large read-only objects by `const&`,
- pass objects by non-`const` reference only when modification is intended.

8.6 Default arguments

A default argument provides a value that will be used if the caller omits that argument.

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

void greet(const std::string& name = "Guest") {
    std::cout << "Hello, " << name << '\n';
}

int main() {
    greet();
    greet("Ayman");
}
```

Output conceptually:

```
Hello, Guest
Hello, Ayman
```

8.6.1 Rules for default arguments

Default arguments follow important rules:

- parameters with defaults must appear at the end of the parameter list,
- default arguments are usually written in the declaration,
- they should not be repeated inconsistently across declarations.

8.6.2 Correct example

```
#include <iostream>

void log_message(int level, bool newline = true) {
    std::cout << "Level: " << level;
    if (newline) {
        std::cout << '\n';
    }
}
```

8.6.3 Incorrect parameter ordering

```
// void f(int x = 10, int y); // invalid
```

This is invalid because a parameter without a default follows one with a default.

8.6.4 Defaults in declarations

A common multi-file pattern is:

```
// printer.hpp
#pragma once

void print_line(int width = 80);
```

```
// printer.cpp
#include <iostream>
#include "printer.hpp"

void print_line(int width) {
    for (int i{0}; i < width; ++i) {
        std::cout << '-';
    }
    std::cout << '\n';
}
```

The default argument appears in the declaration, not repeated in the definition.

8.6.5 When default arguments are useful

Default arguments are useful when:

- one parameter value is the common case,
- you want simpler calls for ordinary usage,
- the omitted argument still has a clear and reasonable meaning.

8.7 Function overloading

Function overloading allows multiple functions to share the same name as long as their parameter lists differ in a valid way.

```
#include <iostream>

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

double add(double a, double b) {
    return a + b;
}

int main() {
    std::cout << add(2, 3) << '\n';
    std::cout << add(2.5, 3.5) << '\n';
}
```

The compiler chooses the appropriate overload based on the argument types.

8.7.1 What can differ in overloads

Functions may be overloaded by:

- different parameter types,
- different numbers of parameters,
- different parameter type ordering.

8.7.2 Different counts

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

int multiply(int a, int b, int c) {
    return a * b * c;
}
```

8.7.3 Return type alone is not enough

You cannot overload functions only by changing the return type.

```
// int compute(int x);
// double compute(int x); // invalid
```

This is invalid because the parameter list is the same.

8.7.4 A practical overload example

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

void print(int value) {
    std::cout << "int: " << value << '\n';
}

void print(double value) {
    std::cout << "double: " << value << '\n';
}

void print(const std::string& value) {
    std::cout << "string: " << value << '\n';
}
```

8.7.5 Overloading and clarity

Overloading is powerful, but overloaded functions should still represent closely related operations. Using the same name for unrelated actions makes interfaces confusing.

8.8 `inline`

The keyword `inline` has an important meaning in C++, but beginners often misunderstand it.

8.8.1 Historical intuition

Historically, `inline` suggested that a function body could be expanded at the call site instead of being called normally.
Example:

```
inline int square(int x) {
    return x * x;
}
```

8.8.2 Modern practical meaning

In modern C++, `inline` is primarily a linkage and definition-management tool. It allows the same function definition to appear in multiple translation units, provided the definitions are identical.
This is especially useful for functions defined in header files.

```cpp
// math.hpp
#pragma once

inline int square(int x) {
    return x * x;
}
```

Now any source file that includes this header can use `square` without causing multiple-definition linker problems, as long as the rules for inline definitions are respected.

8.8.3 The compiler still decides optimization

Even if a function is marked `inline`, the compiler is free to inline or not inline the call as an optimization choice. Therefore:

- `inline` does not force machine-code inlining,
- performance decisions remain under compiler control.

8.8.4 When `inline` is commonly used

`inline` is commonly used for:

- small header-defined utility functions,
- accessor-like functions,
- functions that must be defined in headers and used across multiple translation units.

8.8.5 An example in a header

```cpp
// geometry.hpp
#pragma once

inline int area(int width, int height) {
    return width * height;
}
```

8.8.6 Why `inline` matters for ODR

For a normal non-inline function, there must be one definition in the program. For an inline function, identical definitions may appear in multiple translation units. This is one reason the keyword is important even apart from optimization.

8.9 Forward declarations

A function must be declared before it is used by the compiler in that translation unit.

8.9.1 Basic forward declaration

```
#include <iostream>

int add(int a, int b); // forward declaration

int main() {
    std::cout << add(2, 3) << '\n';
}

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

The declaration tells the compiler:

- the function name,
- the return type,
- the parameter types.

That is enough for the call in `main` to compile, even though the full definition appears later.

8.9.2 Why forward declarations are needed

C++ compilers process each translation unit in a structured way. If a function call appears before the compiler has seen a declaration for that function, the call is not valid.
Incorrect order without declaration:

```
#include <iostream>

int main() {
    std::cout << add(2, 3) << '\n';
}

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

This fails because `add` is unknown at the point of the call.

8.9.3 Forward declarations in header files

In multi-file programs, function declarations are commonly placed in header files.

```
// math.hpp
#pragma once

int add(int a, int b);
int subtract(int a, int b);
```

```cpp
// math.cpp
#include "math.hpp"

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

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

```cpp
// main.cpp
#include <iostream>
#include "math.hpp"

int main() {
    std::cout << add(10, 5) << '\n';
    std::cout << subtract(10, 5) << '\n';
}
```

This is the normal structure for real projects.

8.9.4 Declaration and definition must match

A declaration and its definition must agree in:

- return type,
- function name,
- parameter types,
- namespace placement,
- cv/ref qualifications where relevant.

For example, this would be incorrect:

```cpp
// declaration
int add(int a, int b);

// wrong definition
double add(int a, int b) {
    return a + b;
}
```

8.9.5 Forward declaration versus full definition

A declaration introduces the interface. A definition provides the body.

Declaration only:

```cpp
int multiply(int a, int b);
```

Definition:

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

Conclusion

Functions are one of the most important foundations of C++ program structure. They make code reusable, easier to test, easier to read, and easier to organize across multiple files.
A strong understanding of functions requires more than knowing basic syntax. It includes understanding:

- how definitions and calls work,
- how parameters and return values express data flow,
- when to use pass by value,
- when to use pass by reference,
- why `const&` is a key modern interface tool,
- how default arguments simplify common calls,
- how overloading creates families of related operations,
- what `inline` really means in modern C++,
- why forward declarations are essential for proper program structure.

These ideas affect both correctness and performance. They also shape how interfaces are designed in real C++ code. Once functions are understood well, the programmer is ready to build larger abstractions, organize code across headers and source files, and move toward more advanced features of the language.

Scope and Lifetime

Scope and lifetime are two of the most fundamental concepts in C++. They determine:

- where a name is visible,
- how long an object exists,
- when resources are acquired and released,
- how memory is managed safely.

Many serious bugs in C++ programs are not caused by syntax errors, but by misunderstanding scope and lifetime. A pointer referencing a destroyed object, a static variable shared unintentionally, or a reference outliving its target can lead to undefined behavior that is difficult to detect and debug.
Modern C++ emphasizes writing code that makes object lifetime explicit and safe, relying on scope-based resource management (RAII) and avoiding ambiguous ownership.
This chapter explains:

- different storage durations (local, global, static),
- how scope rules control visibility,
- how object lifetime is determined,
- the meaning of `static` and `thread_local`,
- why lifetime management is critical in real-world C++ programs.

9.1 Local / global / static storage

Every object in C++ has a storage duration, which defines how long it exists in memory. The main categories are:

- automatic storage (local variables),
- static storage (global and static variables),
- thread-local storage.

9.1.1 Local variables (automatic storage)

Local variables are defined inside functions or blocks and have automatic storage duration.

```
#include <iostream>

int main() {
    int x{10}; // local variable
    std::cout << x << '\n';
}
```

Characteristics:

- created when the block is entered,
- destroyed when the block is exited,
- typically stored on the stack,
- lifetime is limited to the block.

9.1.2 Block-local variables

```
int main() {
    {
        int y{20};
    }
    // y is no longer accessible here
}
```

9.1.3 Global variables (static storage)

Global variables are defined outside any function.

```
#include <iostream>

int global_value{100};

int main() {
    std::cout << global_value << '\n';
}
```

Characteristics:

- exist for the entire duration of the program,
- initialized before `main` starts,
- destroyed after `main` ends,
- have static storage duration.

9.1.4 Static storage duration

Objects with static storage duration include:

- global variables,
- static variables,
- static data members,
- variables declared with `static` or `thread_local`.

They exist for the entire program lifetime.

9.2 Scope rules

Scope defines where a name is visible and can be used.

9.2.1 Block scope

A variable declared inside a block is only visible within that block.

```
int main() {
    int x{10};

    {
        int y{20};
        // x and y are visible here
    }

    // y is not visible here
}
```

9.2.2 Function scope

Function parameters and local variables are visible only inside the function.

```
void f(int value) {
    int local{5};
}
```

9.2.3 Global scope

Names declared outside functions are visible throughout the translation unit.

```
int g{10};

int main() {
    return g;
}
```

9.2.4 Name hiding

An inner scope can hide a name from an outer scope.

```
#include <iostream>

int main() {
    int value{5};

    {
        int value{10}; // hides outer value
        std::cout << value << '\n'; // prints 10
    }

    std::cout << value << '\n'; // prints 5
}
```

9.2.5 Scope resolution operator

The global scope operator `::` allows access to global names.

```
#include <iostream>

int value{100};

int main() {
    int value{10};
    std::cout << ::value << '\n'; // global value
}
```

9.3 Object lifetime

Lifetime refers to the period during which an object exists in memory.

9.3.1 Automatic lifetime

Objects with automatic storage duration exist from their declaration until the end of their block.

```
int main() {
    int x{5};
    // x is alive here
}
// x is destroyed here
```

9.3.2 Static lifetime

Objects with static storage duration exist for the entire program.

```
int global_value{10}; // exists for entire program
```

9.3.3 Dynamic lifetime

Objects created with `new` exist until explicitly destroyed.

```
int* p = new int(10);
delete p;
```

Modern C++ prefers avoiding manual memory management in favor of RAII.

9.3.4 Dangling references

A reference or pointer becomes invalid if it refers to an object that no longer exists.

```
int* get_ptr() {
    int x{10};
    return &x; // dangerous
}
```

Here `x` is destroyed when the function returns, leaving a dangling pointer.

9.4 `static`

The keyword `static` has different meanings depending on context.

9.4.1 Static local variables

A static variable inside a function retains its value between calls.

```
#include <iostream>

void counter() {
    static int count{0};
    ++count;
    std::cout << count << '\n';
}

int main() {
    counter();
    counter();
    counter();
}
```

Output conceptually:

```
1
2
3
```

Characteristics:

- initialized once,
- retains value across calls,
- has static storage duration.

9.4.2 Static at global scope

At global scope, `static` limits visibility to the current translation unit.

```
static int internal_value{42};
```

This prevents the symbol from being visible in other source files.

9.4.3 Static member functions and variables

In classes, `static` members belong to the class rather than instances. This will be covered in detail in later chapters.

9.5 `thread_local`

The keyword `thread_local` specifies that each thread has its own instance of a variable.

```
#include <iostream>
#include <thread>

thread_local int counter{0};

void work() {
    ++counter;
    std::cout << counter << '\n';
}

int main() {
    std::thread t1(work);
    std::thread t2(work);

    t1.join();
    t2.join();
}
```

Each thread has its own `counter`.

9.5.1 Characteristics

- each thread gets a separate instance,
- lifetime is tied to the thread,
- useful for thread-specific state.

9.6 Why lifetime matters in C++

Lifetime management is one of the most critical aspects of C++ programming.

9.6.1 Avoiding undefined behavior

Accessing objects outside their lifetime leads to undefined behavior.

```
int* p;

{
    int x{10};
    p = &x;
}
// x is destroyed here

// *p is now invalid
```

9.6.2 RAII principle

Modern C++ uses RAII (Resource Acquisition Is Initialization) to tie resource management to object lifetime.

```
#include <fstream>

int main() {
    std::ofstream file("data.txt");
    // file is automatically closed when it goes out of scope
}
```

9.6.3 Deterministic destruction

Objects are destroyed at well-defined points:

- end of scope for automatic variables,
- program termination for static variables,
- explicit deletion or smart pointer destruction for dynamic objects.

9.6.4 Memory safety

Correct lifetime management prevents:

- dangling pointers,
- use-after-free errors,
- memory leaks,
- data races in multithreaded code.

9.6.5 Modern C++ guideline

Modern C++ encourages:

- avoiding raw pointers for ownership,

- using smart pointers,
- relying on scope-based lifetime,
- making ownership explicit.

Conclusion

Scope and lifetime define how and where objects exist in a C++ program. Understanding them is essential for writing correct, safe, and efficient code.
A professional C++ programmer must understand:

- how local, global, and static storage differ,
- how scope determines visibility,
- how lifetime determines existence,
- how `static` and `thread_local` affect storage,
- why incorrect lifetime handling leads to serious bugs.

Mastering these concepts is a key step toward understanding memory management, resource ownership, and advanced C++ design patterns.

References and Pointers

References and pointers are among the most important concepts in C++, and they are also among the most misunderstood by beginners. They both allow a program to work with objects indirectly, but they do so in different ways and with different rules. A programmer who understands references and pointers properly gains a stronger understanding of:

- how objects are accessed,
- how functions can avoid unnecessary copying,
- how memory addresses are represented,
- why lifetime and ownership matter,
- why Modern C++ does not begin with raw pointers as the default style.

In older teaching material, pointers were often introduced very early and used as the main tool for indirect access and dynamic memory management. Modern C++ still supports pointers fully, and every serious programmer must understand them. However, Modern C++ no longer treats raw pointers as the first or safest abstraction for everyday programming. Instead, references, value semantics, RAII, and smart pointers form the recommended foundation.
This chapter explains:

- what references are,
- what pointers are,
- how they differ,
- what `nullptr` means,
- the memory basics needed to reason about them,
- common pointer mistakes,
- why raw pointers are not the right starting point in Modern C++.

10.1 References

A reference is an alias for an existing object. Once a reference is initialized to refer to an object, it cannot be made to refer to another object later.

10.1.1 Basic reference declaration

```
#include <iostream>

int main() {
    int value{10};
    int& ref = value;

    std::cout << value << '\n';
    std::cout << ref << '\n';
}
```

Here:

- `value` is an `int`,
- `ref` is a reference to `value`,
- both names refer to the same object.

10.1.2 Modifying through a reference

Because a reference refers to the original object, modifying the reference modifies the object itself.

```
#include <iostream>

int main() {
    int value{10};
    int& ref = value;

    ref = 25;

    std::cout << value << '\n'; // prints 25
}
```

10.1.3 A reference must be initialized

A reference is not like an ordinary variable that can exist without referring to something. It must be initialized when declared.

```
// int& ref; // invalid
```

This is invalid because the compiler requires the reference to be bound to an object immediately.

10.1.4 A reference cannot be reseated

Once initialized, a reference remains bound to the same object.

```
#include <iostream>

int main() {
    int a{10};
    int b{20};
```

```
    int& ref = a;
    ref = b;

    std::cout << a << '\n'; // prints 20
    std::cout << b << '\n'; // prints 20
}
```

This does **not** make `ref` refer to `b`. Instead, it assigns the value of `b` to the object already referred to by `ref`, which is `a`.

10.1.5 References in function parameters

References are extremely important in function interfaces.

```
#include <iostream>

void increment(int& x) {
    ++x;
}

int main() {
    int value{5};
    increment(value);

    std::cout << value << '\n'; // prints 6
}
```

This allows a function to modify the caller's object directly.

10.1.6 `const` references

A const reference allows access to an object without copying it and without allowing modification.

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

void print_name(const std::string& name) {
    std::cout << name << '\n';
}
```

This is one of the most common and important patterns in Modern C++.

10.1.7 Reference lifetime caution

A reference must refer to a valid object. If the referred object dies and the reference is still used, the program has a dangling reference.

```
int& bad_reference() {
    int local{42};
    return local; // dangerous
}
```

The local variable is destroyed when the function ends, so the returned reference becomes invalid.

10.2 Pointers

A pointer is an object that stores the address of another object or function. Unlike a reference, a pointer is itself a separate object with its own value.

10.2.1 Basic pointer declaration

```
#include <iostream>

int main() {
    int value{10};
    int* ptr = &value;

    std::cout << value << '\n';
    std::cout << *ptr << '\n';
}
```

Here:

- `&value` means "the address of `value`",
- `ptr` stores that address,
- `*ptr` means "the object pointed to by `ptr`".

10.2.2 Address-of and indirection

Two operators are fundamental:

- `&` gives the address of an object,
- `*` dereferences a pointer to access the pointed-to object.

```
int value{7};
int* ptr = &value;

int copy = *ptr;
```

10.2.3 Changing an object through a pointer

```
#include <iostream>

int main() {
    int value{10};
    int* ptr = &value;

    *ptr = 99;

    std::cout << value << '\n'; // prints 99
}
```

10.2.4 Pointers can be reassigned

Unlike references, pointers can be changed to point somewhere else.

```
#include <iostream>

int main() {
    int a{10};
    int b{20};

    int* ptr = &a;
    std::cout << *ptr << '\n'; // 10

    ptr = &b;
    std::cout << *ptr << '\n'; // 20
}
```

10.2.5 Pointers can be null

A pointer can intentionally point to nothing.

```
int* ptr = nullptr;
```

This is one of the major differences between pointers and references.

10.2.6 Pointers to dynamic objects

Pointers are often associated with dynamic allocation:

```
int* ptr = new int(42);
delete ptr;
```

This is valid C++, but Modern C++ strongly discourages using raw owning pointers as a beginner's default style. Later chapters will show why smart pointers are usually better.

10.2.7 Pointers are objects too

A pointer itself has storage, a type, and a value.

```
int value{5};
int* ptr = &value;
```

Here:

- `value` is an `int` object,
- `ptr` is an `int*` object,
- the value stored in `ptr` is an address.

10.3 Differences between them

References and pointers are related, but they are not interchangeable.

10.3.1 Reference versus pointer summary

A reference:

- must be initialized,
- cannot be null in ordinary usage,
- cannot be reseated,
- behaves like another name for an object.

A pointer:

- may be uninitialized if written carelessly,
- may be null,
- may be reseated,
- is a separate object that stores an address.

10.3.2 Syntax difference

Reference:

```
int value{10};
int& ref = value;
```

Pointer:

```
int value{10};
int* ptr = &value;
```

10.3.3 Usage difference

Reference access:

```
ref = 20;
```

Pointer access:

```
*ptr = 20;
```

10.3.4 Can it be null?

Reference:

```
// int& ref = ??? // must bind to something valid
```

Pointer:

```
int* ptr = nullptr;
```

10.3.5 Can it be reseated?

Reference:

```
int a{1};
int b{2};
int& ref = a;

// ref = b; // assigns b's value to a
```

Pointer:

```
int a{1};
int b{2};
int* ptr = &a;

ptr = &b; // now points to b
```

10.3.6 When references are preferred

References are usually preferred when:

- null is not a meaningful state,
- the parameter must refer to a valid object,
- the goal is simply aliasing or efficient argument passing.

10.3.7 When pointers are appropriate

Pointers are appropriate when:

- null is a meaningful value,
- reseating is needed,
- pointer arithmetic or low-level memory traversal is required,
- interfacing with low-level APIs or data structures requires them.

10.4 `nullptr`

Modern C++ uses `nullptr` to represent the null pointer value.

10.4.1 Basic meaning

```
int* ptr = nullptr;
```

This means `ptr` does not currently point to any object.

10.4.2 Why `nullptr` exists

Before C++11, programmers often used `0` or `NULL`. That was error-prone because those forms interacted badly with overload resolution and integer semantics.
Modern C++ uses `nullptr`, which has the dedicated type `std::nullptr_t`.

10.4.3 Example of safer overload behavior

```
#include <iostream>

void f(int) {
    std::cout << "int overload\n";
}

void f(int*) {
    std::cout << "pointer overload\n";
}

int main() {
    f(nullptr); // calls pointer overload
}
```

This is one of the reasons `nullptr` is better than `0`.

10.4.4 Checking for null

```
#include <iostream>

int main() {
    int* ptr = nullptr;

    if (ptr == nullptr) {
        std::cout << "Pointer is null\n";
    }
}
```

Or simply:

```
if (!ptr) {
    std::cout << "Pointer is null\n";
}
```

10.4.5 Never dereference `nullptr`

This is invalid:

```
int* ptr = nullptr;
// *ptr = 5; // undefined behavior
```

A null pointer does not point to an object.

10.5 Memory basics

To understand pointers, a beginner needs a simple but correct view of memory.

10.5.1 Objects occupy storage

Every object in a C++ program occupies storage. A pointer stores the address of an object's storage.

```
int value{42};
int* ptr = &value;
```

You can think of this as:

- `value` is an object,
- it lives somewhere in memory,
- `ptr` stores where that object is.

10.5.2 Automatic storage

Many objects are created automatically when execution enters a block and destroyed when the block ends.

```
int main() {
    int local{10};
}
```

`local` exists only until the end of the block.

10.5.3 Dynamic storage

Objects created with `new` have dynamic storage duration.

```
int* ptr = new int(10);
delete ptr;
```

The storage persists until explicitly released, or until a smart pointer releases it automatically.

10.5.4 Lifetime matters more than just address

A pointer is only useful if the object it points to is still alive.

```
int* bad_ptr() {
    int local{42};
    return &local; // dangerous
}
```

The pointer may still contain an address-like value, but the object no longer exists after the function returns.

10.5.5 Memory is not ownership

A raw pointer only stores an address. It does **not** automatically manage the lifetime of the object it points to.

```
int* ptr = new int(5);
```

This allocates memory, but nothing in the raw pointer itself guarantees that `delete` will happen correctly.
That is one reason Modern C++ emphasizes ownership-managing abstractions such as `std::unique_ptr`.

10.6 Common pointer mistakes

Pointer mistakes are among the most common sources of serious C++ bugs.

10.6.1 Uninitialized pointers

A raw pointer that is not initialized contains an indeterminate value.

```
int* ptr; // dangerous
```

Dereferencing such a pointer is undefined behavior.
Safer:

```
int* ptr = nullptr;
```

10.6.2 Dereferencing null

```
int* ptr = nullptr;
// std::cout << *ptr; // undefined behavior
```

Always check or ensure validity before dereferencing.

10.6.3 Dangling pointers

A pointer dangles when the object it pointed to no longer exists.

```
int* ptr{};

{
    int value{10};
    ptr = &value;
}

// ptr now dangles
```

10.6.4 Memory leaks

A memory leak occurs when dynamically allocated memory is never released.

```
void leak() {
    int* ptr = new int(42);
}
```

When the function ends, the pointer is lost but the allocated object remains unreleased.

10.6.5 Double delete

Deleting the same dynamic object more than once is undefined behavior.

```
int* ptr = new int(5);
delete ptr;
// delete ptr; // undefined behavior
```

10.6.6 Using deleted memory

Even if a pointer still contains an address, the pointed object is gone after deletion.

```
int* ptr = new int(10);
delete ptr;

// *ptr = 20; // undefined behavior
```

A common defensive step is:

```
int* ptr = new int(10);
delete ptr;
ptr = nullptr;
```

This does not solve ownership design, but it reduces one class of bug in simple code.

10.6.7 Confusing ownership with observation

A raw pointer may sometimes mean:

- “I own this object and must delete it,”
- or “I am only observing an object owned elsewhere.”

If the program does not make that distinction clear, bugs become likely.

10.7 Why raw pointers are not the starting point in Modern C++

Raw pointers are still part of C++, and every serious programmer must understand them. But Modern C++ no longer teaches them as the primary default tool for ownership and routine object management.

10.7.1 Modern C++ begins with values, references, and RAII

A beginner should first become comfortable with:

- ordinary objects,
- automatic storage,
- references,
- const references,

- standard containers,
- RAII.

These ideas produce code that is safer and easier to reason about.

10.7.2 Manual `new`/`delete` is error-prone

This style is legal but fragile:

```
int* ptr = new int(42);
delete ptr;
```

It introduces many risks:

- forgetting to delete,
- deleting twice,
- using the pointer after deletion,
- losing the pointer and leaking memory.

10.7.3 RAII-based alternatives are better

Modern C++ prefers ownership-managing objects.

```
#include <memory>

int main() {
    auto ptr = std::make_unique<int>(42);
}
```

This still uses a pointer conceptually, but the lifetime is managed automatically.

10.7.4 Raw pointers still matter

Raw pointers are still important for:

- low-level code,
- interoperability with C APIs,
- non-owning observation,
- arrays, buffers, and memory traversal,
- systems programming and data-structure implementation.

So the goal is not to avoid learning raw pointers. The goal is to learn them in the right context.

10.7.5 A better beginner mindset

Instead of thinking:

> "Whenever I need indirect access, I should start with raw pointers."

Modern C++ encourages this mindset:

> "Start with values and references. Use raw pointers only when their semantics are truly needed. Use smart pointers when ownership of dynamic objects must be managed."

10.7.6 A practical comparison

Older style:

```cpp
#include <iostream>

int main() {
    int* ptr = new int(100);
    std::cout << *ptr << '\n';
    delete ptr;
}
```

Modern style:

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

int main() {
    auto ptr = std::make_unique<int>(100);
    std::cout << *ptr << '\n';
}
```

The second version expresses ownership much more safely.

Conclusion

References and pointers both provide indirect access to objects, but they serve different purposes.
References are best understood as aliases:

- they must be initialized,
- they cannot be null in ordinary use,
- they cannot be reseated,
- they are ideal for many function interfaces.

Pointers are more flexible but also more dangerous:

- they store addresses,

- they may be null,
- they may be reseated,
- they can participate in low-level memory operations,
- they require greater care because they do not manage lifetime by themselves.

Modern C++ still requires a strong understanding of raw pointers, but it does not begin with them as the default tool for ownership and everyday programming. The safer starting point is value semantics, references, const references, RAII, and later smart pointers.
That is the modern path: understand raw pointers deeply, but do not make them your first everyday abstraction.

Part IV

Arrays, Strings, and Containers

Arrays and Modern Alternatives

Arrays are one of the oldest and most fundamental ways to store a sequence of elements in programming. C++ still supports traditional C-style arrays, and every serious C++ programmer must understand them. However, Modern C++ no longer treats raw built-in arrays as the default tool for most everyday work.
The reason is simple: while C-style arrays are fast and map directly to contiguous memory, they have important limitations. They do not know their own size in most function interfaces, they decay to pointers in many expressions, and they are easy to misuse. Modern C++ therefore provides safer and clearer alternatives such as `std::array` and `std::span`, which preserve the benefits of contiguous storage while improving correctness and expressiveness.
This chapter explains:

- what C-style arrays are,
- what their limitations are,
- how `std::array` improves fixed-size array usage,
- how `std::span` provides a non-owning view over contiguous memory,
- when to use each representation,
- how to handle contiguous data safely in Modern C++.

11.1 C-style arrays

A C-style array is a fixed-size sequence of elements of the same type stored contiguously in memory.

11.1.1 Basic declaration

```
#include <iostream>

int main() {
    int values[5]{10, 20, 30, 40, 50};

    for (int i{0}; i < 5; ++i) {
        std::cout << values[i] << '\n';
    }
}
```

In this example:

- `values` is an array of five `int` objects,
- the elements are stored one after another in contiguous memory,
- indexing starts at zero.

11.1.2 Array size is part of the type

For built-in arrays, the size is part of the type.

```
int a[3]{1, 2, 3};
int b[5]{1, 2, 3, 4, 5};
```

These are different types:

- `a` has type “array of 3 int”,
- `b` has type “array of 5 int”.

11.1.3 Contiguous storage

Array elements are laid out contiguously, which means the next element follows immediately in memory.

```
#include <iostream>

int main() {
    int values[4]{10, 20, 30, 40};

    std::cout << values[0] << '\n';
    std::cout << values[1] << '\n';
    std::cout << values[2] << '\n';
    std::cout << values[3] << '\n';
}
```

This contiguous layout is one reason arrays can be efficient and compatible with low-level APIs.

11.1.4 Default indexing syntax

Elements are accessed with the subscript operator:

```
int data[3]{7, 8, 9};

int first = data[0];
int second = data[1];
int third = data[2];
```

11.1.5 Modification

Elements may be modified directly:

```cpp
#include <iostream>

int main() {
    int values[3]{1, 2, 3};

    values[1] = 99;

    for (int i{0}; i < 3; ++i) {
        std::cout << values[i] << '\n';
    }
}
```

11.1.6 Partial initialization

If fewer initializers are provided than the array size, the remaining elements are value-initialized.

```cpp
#include <iostream>

int main() {
    int values[5]{1, 2};

    for (int i{0}; i < 5; ++i) {
        std::cout << values[i] << '\n';
    }
}
```

This produces:

- `values[0] == 1`
- `values[1] == 2`
- the rest become zero

11.1.7 Character arrays

C-style arrays are also commonly used for character data, especially in older code.

```cpp
#include <iostream>

int main() {
    char text[6]{'H', 'e', 'l', 'l', 'o', '\0'};
    std::cout << text << '\n';
}
```

However, for text handling in Modern C++, `std::string` is usually a better default.

11.1.8 Multidimensional arrays

Built-in arrays can be multidimensional:

```
#include <iostream>

int main() {
    int matrix[2][3]{
        {1, 2, 3},
        {4, 5, 6}
    };

    for (int row{0}; row < 2; ++row) {
        for (int col{0}; col < 3; ++col) {
            std::cout << matrix[row][col] << ' ';
        }
        std::cout << '\n';
    }
}
```

This remains useful in some low-level and performance-sensitive contexts.

11.2 Limitations

Although C-style arrays are fundamental and efficient, they have several limitations that make them less safe and less expressive than modern alternatives.

11.2.1 They do not know their size in most interfaces

One of the biggest practical problems is that arrays decay to pointers when passed to functions.

```
#include <iostream>

void print_values(int values[]) {
    // values is treated as int*
    std::cout << values[0] << '\n';
}

int main() {
    int data[3]{10, 20, 30};
    print_values(data);
}
```

Inside the function, `values` is no longer a real array type. It is effectively a pointer to the first element, so the function has no built-in knowledge of the number of elements.

11.2.2 Size must often be passed separately

Because of that decay behavior, a separate size parameter is often needed.

```
#include <iostream>

void print_values(const int* values, int count) {
    for (int i{0}; i < count; ++i) {
```

```
        std::cout << values[i] << '\n';
    }
}

int main() {
    int data[4]{5, 10, 15, 20};
    print_values(data, 4);
}
```

This works, but now correctness depends on the caller passing the right pointer and the right size together.

11.2.3 No bounds checking with `operator[]`

Built-in array indexing does not perform bounds checking.

```
int values[3]{1, 2, 3};

// values[10] = 99; // undefined behavior
```

The language does not stop this. Accessing beyond the valid range leads to undefined behavior.

11.2.4 Arrays cannot be copied by assignment

Built-in arrays do not support ordinary assignment as complete value objects.

```
int a[3]{1, 2, 3};
int b[3]{4, 5, 6};

// a = b; // invalid
```

This is one reason they are less convenient than standard library containers.

11.2.5 Awkward interaction with function interfaces

Because arrays decay to pointers in many expressions, they lose some important type information at exactly the places where interface safety matters most.

```
void process(int data[100]);
```

This does not force the caller to provide an array of 100 elements. In a function parameter list, it is treated as a pointer parameter.

11.2.6 Difficult to return directly by value

Returning a built-in array from a function is not supported in the same convenient way as returning a standard library object. This encourages more awkward designs in older code.

11.2.7 Pointer decay causes subtle bugs

Consider:

```
#include <iostream>

void show_size(int values[]) {
    std::cout << sizeof(values) << '\n';
}

int main() {
    int data[5]{};
    std::cout << sizeof(data) << '\n';
    show_size(data);
}
```

In `main`, `sizeof(data)` gives the size of the entire array object. Inside `show_size`, `values` is treated as a pointer parameter, so the result is not the full array size.

11.2.8 Not the best default for modern everyday code

C-style arrays remain important for interoperability, low-level memory work, compile-time fixed data, and language understanding. But for many ordinary tasks, Modern C++ prefers more expressive and safer abstractions.

11.3 `std::array`

`std::array` is a Standard Library container that represents a fixed-size sequence of elements stored contiguously.
It keeps the "fixed-size contiguous storage" idea of built-in arrays, but behaves much more like a normal value type.

11.3.1 Basic usage

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

int main() {
    std::array<int, 5> values{10, 20, 30, 40, 50};

    for (std::size_t i{0}; i < values.size(); ++i) {
        std::cout << values[i] << '\n';
    }
}
```

11.3.2 Why `std::array` is better than a C-style array

`std::array` improves fixed-size array programming because:

- it knows its own size,
- it supports ordinary copying and assignment,

- it integrates naturally with algorithms and range-based loops,
- it offers member functions such as `size()`, `data()`, `front()`, and `back()`.

11.3.3 Size is part of the type and easy to query

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

int main() {
    std::array<int, 4> values{1, 2, 3, 4};
    std::cout << values.size() << '\n';
}
```

This is much clearer than manually computing array size with `sizeof` arithmetic.

11.3.4 Copying and assignment

Unlike built-in arrays, `std::array` supports ordinary value semantics.

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

int main() {
    std::array<int, 3> a{1, 2, 3};
    std::array<int, 3> b{4, 5, 6};

    a = b;

    for (int value : a) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

11.3.5 Range-based iteration

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

int main() {
    std::array<int, 4> values{5, 10, 15, 20};

    for (int value : values) {
        std::cout << value << '\n';
    }
}
```

11.3.6 Access with `at()`

In addition to `operator[]`, `std::array` provides `at()`, which performs bounds checking and throws an exception if the index is invalid.

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

int main() {
    std::array<int, 3> values{10, 20, 30};

    try {
        std::cout << values.at(1) << '\n';
        std::cout << values.at(5) << '\n';
    } catch (const std::out_of_range& e) {
        std::cout << "Out of range access detected\n";
    }
}
```

11.3.7 Pointer access with `data()`

`std::array` stores elements contiguously, so it can still interoperate with low-level code through `data()`.

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

int main() {
    std::array<int, 3> values{7, 8, 9};
    int* raw = values.data();

    std::cout << raw[0] << '\n';
    std::cout << raw[1] << '\n';
    std::cout << raw[2] << '\n';
}
```

11.3.8 A function taking `std::array`

Because `std::array` preserves size information, interfaces can be clearer.

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

int sum(const std::array<int, 4>& values) {
    int total{0};

    for (int value : values) {
        total += value;
    }

    return total;
}
```

```
int main() {
    std::array<int, 4> data{1, 2, 3, 4};
    std::cout << sum(data) << '\n';
}
```

11.3.9 A compile-time fixed buffer example

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

int main() {
    std::array<char, 6> text{'H', 'e', 'l', 'l', 'o', '\0'};
    std::cout << text.data() << '\n';
}
```

11.4 `std::span`

`std::span` is one of the most important modern tools for working with contiguous data safely and efficiently.
A span is a lightweight non-owning view over a contiguous sequence of objects. It does not own the elements. Instead, it refers to existing contiguous storage such as:

- a built-in array,
- a `std::array`,
- a `std::vector`,
- another contiguous buffer.

11.4.1 Basic idea

```
#include <iostream>
#include <span>

void print_values(std::span<const int> values) {
    for (int value : values) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}

int main() {
    int raw[]{10, 20, 30, 40};
    print_values(raw);
}
```

This is one of the major advantages of `std::span`: it allows a function to accept contiguous data without losing the size information the way plain pointer parameters do.

11.4.2 A span does not own data

This is essential. A span is only a view. The underlying data must remain alive while the span is used.

```
#include <span>
#include <vector>

int main() {
    std::vector<int> values{1, 2, 3, 4};
    std::span<int> view{values};

    view[0] = 99;
}
```

Modifying the span modifies the original data because the span refers to the same storage.

11.4.3 Constructing from a built-in array

```
#include <iostream>
#include <span>

int main() {
    int values[]{5, 10, 15, 20};
    std::span<int> s{values};

    for (int value : s) {
        std::cout << value << '\n';
    }
}
```

11.4.4 Constructing from `std::array`

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

int main() {
    std::array<int, 4> values{1, 2, 3, 4};
    std::span<int> s{values};

    for (int value : s) {
        std::cout << value << '\n';
    }
}
```

11.4.5 Constructing from `std::vector`

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

int main() {
```

```
    std::vector<int> values{100, 200, 300};
    std::span<int> s{values};

    for (int value : s) {
        std::cout << value << '\n';
    }
}
```

11.4.6 Const span for read-only access

A very common interface form is `std::span<const T>`.

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

void print_values(std::span<const int> values) {
    for (int value : values) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}

int main() {
    std::vector<int> data{2, 4, 6, 8};
    print_values(data);
}
```

This avoids copying and clearly communicates read-only intent.

11.4.7 Subspans

A span can represent a portion of a contiguous sequence.

```
#include <iostream>
#include <span>

int main() {
    int values[]{10, 20, 30, 40, 50};
    std::span<int> all{values};
    std::span<int> part = all.subspan(1, 3);

    for (int value : part) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

11.4.8 First and last elements as views

```
#include <iostream>
#include <span>
```

```
int main() {
    int values[]{1, 2, 3, 4, 5, 6};
    std::span<int> s{values};

    auto first_three = s.first<3>();
    auto last_two = s.last<2>();

    for (int value : first_three) {
        std::cout << value << ' ';
    }
    std::cout << '\n';

    for (int value : last_two) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

11.4.9 Why `std::span` matters

`std::span` solves a long-standing interface problem in C++:

- pointers plus size were error-prone,
- built-in arrays decayed to pointers,
- different container types needed different overloads.

With `std::span`, a function can often take one clean contiguous-view interface.

11.4.10 A modern processing function

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

int sum(std::span<const int> values) {
    int total{0};

    for (int value : values) {
        total += value;
    }

    return total;
}

int main() {
    int raw[]{1, 2, 3};
    std::vector<int> vec{4, 5, 6};

    std::cout << sum(raw) << '\n';
```

```
    std::cout << sum(vec) << '\n';
}
```

11.5 When to use each

Modern C++ does not say that one representation should replace all others. The right choice depends on ownership, size knowledge, mutability, and interface needs.

11.5.1 Use C-style arrays when

Built-in arrays are still appropriate when:

- interfacing with low-level C APIs,
- teaching core language behavior,
- working close to the hardware or ABI boundary,
- using fixed compile-time arrays in limited low-level contexts.

Example:

```
char buffer[256]{};
```

Even here, they should be used carefully.

11.5.2 Use `std::array` when

Use `std::array` when:

- the size is fixed at compile time,
- the object should behave like a normal value type,
- you want container-style operations such as `size()`, `begin()`, and `end()`,
- you want safer and clearer fixed-size storage than a built-in array.

Example:

```
std::array<int, 8> counters{};
```

This is often the best fixed-size choice in Modern C++.

11.5.3 Use `std::span` when

Use `std::span` when:

- a function needs a view over existing contiguous data,
- ownership should remain elsewhere,
- the interface should accept built-in arrays, `std::array`, or `std::vector` uniformly,

- you want to avoid passing pointer-plus-size manually.

Example:

```
void process(std::span<const int> values);
```

11.5.4 Use `std::vector` when size is dynamic

Although this chapter focuses on arrays and nearby alternatives, it is important to state the practical rule clearly:

- use `std::vector` when the number of elements changes at run time or is not known at compile time.

That is one reason Microsoft explicitly recommends `std::vector` or `std::array` over C-style arrays in modern code.

11.5.5 A useful mental model

A practical decision guide is:

- **owning, fixed size:** `std::array`
- **owning, dynamic size:** `std::vector`
- **non-owning contiguous view:** `std::span`
- **low-level legacy or ABI boundary:** built-in array or raw pointer form when necessary

11.6 Safe handling of contiguous data

The central goal of Modern C++ is not to eliminate contiguous storage, but to handle it more safely and clearly.

11.6.1 Prefer abstractions that preserve size information

Unsafe traditional interface:

```
#include <iostream>

void print_values(const int* data, std::size_t count) {
    for (std::size_t i{0}; i < count; ++i) {
        std::cout << data[i] << ' ';
    }
    std::cout << '\n';
}
```

This works, but correctness depends on two separate values remaining consistent.
Safer modern interface:

```
#include <iostream>
#include <span>

void print_values(std::span<const int> data) {
    for (int value : data) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

11.6.2 Prefer range-based loops

Range-based loops reduce indexing mistakes when the index itself is unnecessary.

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

int main() {
    std::array<int, 5> values{10, 20, 30, 40, 50};

    for (int value : values) {
        std::cout << value << '\n';
    }
}
```

11.6.3 Use bounds-checked access when correctness matters

If bounds safety matters more than the tiny cost of checking, prefer `at()` where available.

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

int main() {
    std::array<int, 3> values{1, 2, 3};

    try {
        std::cout << values.at(2) << '\n';
        std::cout << values.at(10) << '\n';
    } catch (const std::out_of_range&) {
        std::cout << "Invalid index detected\n";
    }
}
```

11.6.4 Do not construct spans from dead data

A span does not extend lifetime. This is critical.

Dangerous idea:

```
#include <span>
#include <vector>

std::span<int> bad_view() {
    std::vector<int> values{1, 2, 3};
    return std::span<int>{values}; // dangerous
}
```

The vector is destroyed when the function ends, so the returned span becomes invalid.

11.6.5 Use const views for read-only processing

If a function does not need to modify elements, prefer `std::span<const T>`.

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

void show(std::span<const int> values) {
    for (int value : values) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

This communicates intent and prevents accidental modification.

11.6.6 Prefer `data()` only when you truly need raw access

Even with modern containers, raw pointers can still be obtained when interoperability requires them.

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

void low_level_use(const int* ptr, std::size_t count) {
    for (std::size_t i{0}; i < count; ++i) {
        std::cout << ptr[i] << ' ';
    }
    std::cout << '\n';
}

int main() {
    std::array<int, 4> values{11, 22, 33, 44};
    low_level_use(values.data(), values.size());
}
```

This is appropriate when a low-level interface requires raw pointer access, but it should not be the first design choice for normal high-level code.

11.6.7 A modern end-to-end example

```
#include <array>
#include <iostream>
#include <span>
#include <vector>

void multiply_by_two(std::span<int> values) {
    for (int& value : values) {
        value *= 2;
    }
}

void print_values(std::span<const int> values) {
    for (int value : values) {
        std::cout << value << ' ';
    }
```

```
    std::cout << '\n';
}

int main() {
    std::array<int, 4> fixed{1, 2, 3, 4};
    std::vector<int> dynamic{5, 6, 7, 8};

    multiply_by_two(fixed);
    multiply_by_two(dynamic);

    print_values(fixed);
    print_values(dynamic);
}
```

This example shows a very modern style:

- `std::array` owns fixed-size storage,
- `std::vector` owns dynamic-size storage,
- `std::span` provides a common non-owning contiguous view,
- one interface works with both.

Conclusion

C-style arrays are a core part of the C++ language, and every serious programmer must understand how they work. They provide fixed-size contiguous storage and remain important for low-level code, legacy interoperability, and language fundamentals. However, they also have important limitations:

- they decay to pointers in many expressions,
- they lose size information in common interfaces,
- they do not support normal value semantics,
- they are easy to misuse.

Modern C++ therefore encourages safer alternatives.
`std::array` is the preferred fixed-size owning container when the number of elements is known at compile time. It retains contiguous storage while providing a more complete container interface.
`std::span` is the preferred non-owning view when a function needs to operate on contiguous data without copying it and without taking ownership.
The practical Modern C++ mindset is clear:

- use built-in arrays when low-level language-level behavior is specifically needed,
- use `std::array` for fixed-size owning data,
- use `std::span` for safe non-owning views over contiguous sequences,
- preserve size information and lifetime correctness whenever possible.

That is how Modern C++ keeps the performance advantages of contiguous memory while moving toward clearer and safer code.

Strings

Strings are one of the most commonly used abstractions in programming. In C++, string handling has evolved significantly from raw character arrays to rich, safe, and expressive abstractions in the Standard Library.
Modern C++ provides two primary tools for working with textual data:

- `std::string` for owning and managing string data,
- `std::string_view` for lightweight, non-owning views.

Understanding the difference between owning and non-owning string types is essential for writing efficient and correct programs.
This chapter explains:

- how `std::string` works,
- how `std::string_view` works,
- how to build and modify strings,
- how to parse and convert data,
- the basics of character encoding,
- when to use owning versus non-owning string representations.

12.1 `std::string`

`std::string` is the Standard Library type used to represent a sequence of characters. It manages its own memory and behaves like a dynamic container.

12.1.1 Basic usage

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

int main() {
    std::string text = "Hello, Modern C++";
    std::cout << text << '\n';
}
```

12.1.2 Properties

`std::string`:

- owns its data,
- manages memory automatically,
- supports dynamic resizing,
- stores characters contiguously,
- integrates with standard algorithms.

12.1.3 Size and capacity

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

int main() {
    std::string text = "Hello";

    std::cout << text.size() << '\n';
    std::cout << text.capacity() << '\n';
}
```

12.1.4 Accessing characters

```
#include <string>

int main() {
    std::string text = "ABCDE";

    char first = text[0];
    char second = text.at(1);
}
```

`at()` performs bounds checking, while `operator[]` does not.

12.1.5 Contiguous storage

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

int main() {
    std::string text = "Data";
    const char* ptr = text.data();

    std::cout << ptr << '\n';
}
```

12.2 `std::string_view`

`std::string_view` is a lightweight, non-owning view of a character sequence.

12.2.1 Basic usage

```
#include <iostream>
#include <string_view>

int main() {
    std::string_view view = "Hello, World";
    std::cout << view << '\n';
}
```

12.2.2 Key characteristics

- does not own the data,
- does not allocate memory,
- stores a pointer and a length,
- very cheap to copy,
- ideal for read-only access.

12.2.3 From `std::string`

```
#include <string>
#include <string_view>

int main() {
    std::string text = "Modern C++";
    std::string_view view = text;
}
```

12.2.4 Substrings without allocation

```
#include <iostream>
#include <string_view>

int main() {
    std::string_view text = "Hello, World";

    std::string_view sub = text.substr(7, 5);
    std::cout << sub << '\n';
}
```

12.2.5 Lifetime caution

```
#include <string>
#include <string_view>

std::string_view bad() {
    std::string temp = "Hello";
    return temp; // dangerous
}
```

The returned view becomes invalid because the string is destroyed.

12.3 Building and modifying strings

12.3.1 Concatenation

```
#include <string>

int main() {
    std::string a = "Hello";
    std::string b = "World";

    std::string result = a + " " + b;
}
```

12.3.2 Appending

```
#include <string>

int main() {
    std::string text = "Hello";
    text += " C++";
}
```

12.3.3 Using append

```
#include <string>

int main() {
    std::string text = "Start";
    text.append(" Middle");
    text.append(" End");
}
```

12.3.4 Inserting

```
#include <string>

int main() {
    std::string text = "Hello World";
    text.insert(5, ",");
```

```
}
```

12.3.5 Erasing

```
#include <string>

int main() {
    std::string text = "Hello, World";
    text.erase(5, 1);
}
```

12.3.6 Replacing

```
#include <string>

int main() {
    std::string text = "Hello World";
    text.replace(6, 5, "C++");
}
```

12.3.7 Iterating over characters

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

int main() {
    std::string text = "ABC";

    for (char c : text) {
        std::cout << c << '\n';
    }
}
```

12.4 Parsing and conversions

Modern C++ provides robust utilities for converting between strings and numeric values.

12.4.1 String to integer

```
#include <string>

int main() {
    std::string text = "123";
    int value = std::stoi(text);
}
```

12.4.2 String to floating point

```
#include <string>
```

```cpp
int main() {
    std::string text = "3.14";
    double value = std::stod(text);
}
```

12.4.3 Number to string

```cpp
#include <string>

int main() {
    int value = 42;
    std::string text = std::to_string(value);
}
```

12.4.4 Modern parsing with `from_chars`

```cpp
#include <charconv>
#include <iostream>

int main() {
    const char* str = "123";
    int value{};

    auto result = std::from_chars(str, str + 3, value);

    if (result.ec == std::errc()) {
        std::cout << value << '\n';
    }
}
```

12.4.5 Modern formatting with `to_chars`

```cpp
#include <charconv>
#include <iostream>

int main() {
    char buffer[20];
    int value = 456;

    auto result = std::to_chars(buffer, buffer + 20, value);

    std::cout.write(buffer, result.ptr - buffer);
}
```

12.5 Encoding basics

C++ supports multiple character types to represent different encodings.

12.5.1 Character types

- `char` (usually UTF-8 or implementation-defined)
- `char8_t` (UTF-8)
- `char16_t` (UTF-16)
- `char32_t` (UTF-32)
- `wchar_t` (platform-dependent)

12.5.2 UTF-8 string literal

```
#include <iostream>

int main() {
    const char8_t* text = u8"Hello";
}
```

12.5.3 Unicode string types

```
#include <iostream>

int main() {
    const char16_t* text16 = u"Hello";
    const char32_t* text32 = U"Hello";
}
```

12.5.4 Encoding awareness

Modern C++ does not automatically interpret encoding semantics. A `std::string` is simply a sequence of bytes. It is the programmer's responsibility to ensure correct encoding interpretation.

12.6 Owning vs non-owning strings

Understanding ownership is essential for writing correct and efficient programs.

12.6.1 Owning strings

`std::string` owns its data.

```
#include <string>

int main() {
    std::string text = "Owned string";
}
```

Characteristics:

- manages memory,

- safe to return from functions,
- safe to store long-term.

12.6.2 Non-owning strings

`std::string_view` does not own data.

```
#include <string_view>

int main() {
    std::string_view view = "Temporary view";
}
```

Characteristics:

- lightweight,
- no allocation,
- must not outlive the referenced data.

12.6.3 Choosing between them

Use `std::string` when:

- ownership is required,
- data must persist independently,
- modification is needed.

Use `std::string_view` when:

- read-only access is sufficient,
- no ownership is needed,
- performance matters,
- avoiding copies is important.

12.6.4 A modern function interface

```
#include <iostream>
#include <string_view>

void print(std::string_view text) {
    std::cout << text << '\n';
}

int main() {
    std::string s = "Hello";
```

```
    print(s);
    print("World");
}
```

This allows flexible and efficient usage.

Conclusion

Modern C++ provides powerful and flexible tools for handling strings. The combination of `std::string` and `std::string_view` allows programs to balance safety, performance, and clarity.
Key ideas:

- `std::string` owns and manages memory,
- `std::string_view` provides efficient non-owning access,
- modern parsing and formatting APIs improve performance,
- encoding must be handled carefully,
- choosing the right abstraction depends on ownership and lifetime.

Mastering these tools is essential for writing modern, efficient, and correct C++ programs.

Standard Containers

Standard containers are one of the greatest strengths of Modern C++. They provide well-tested, reusable, efficient data structures that solve common storage and retrieval problems without forcing the programmer to reinvent them manually.
In low-level or early-stage programming, beginners are often tempted to think only in terms of raw arrays, pointers, and hand-written data structures. However, Modern C++ encourages a different mindset. Instead of starting from raw storage and building every structure manually, the programmer should first understand the standard containers and choose the one whose semantics best match the problem.
A container is more than a place to store values. It defines:

- how elements are organized,
- how elements are accessed,
- how insertion and removal behave,
- how iteration works,
- what performance characteristics are typical.

The Standard Library provides several important container families:

- sequence containers, such as `std::vector`, `std::deque`, and `std::list`,
- ordered associative containers, such as `std::map` and `std::set`,
- unordered associative containers, such as `std::unordered_map` and `std::unordered_set`.

This chapter explains why containers matter, how the most important standard containers work, how to choose between them, and how to build a practical intuition for their complexity and usage.

13.1 Why containers matter

Containers matter because almost every real program must store and organize collections of values.
Without standard containers, the programmer would need to manually manage:

- memory allocation,
- resizing,
- insertion and removal logic,

- iteration support,
- searching,
- sorting integration,
- correctness under modification.

That approach is error-prone and wastes time.
Modern C++ containers matter for several reasons.

13.1.1 They encode intent

A container choice communicates meaning.
For example:

- `std::vector<int>` suggests a resizable sequence with fast random access,
- `std::map<std::string, int>` suggests key-value lookup in sorted key order,
- `std::unordered_set<std::string>` suggests fast membership testing for unique strings.

The container itself becomes part of the program's design language.

13.1.2 They reduce bugs

A standard container manages its own internal storage and invariants. That reduces many classes of bugs related to:

- memory leaks,
- invalid manual resizing,
- incorrect pointer arithmetic,
- hand-written linked-structure mistakes,
- size bookkeeping errors.

13.1.3 They integrate with algorithms and ranges

The Standard Library is designed so containers work naturally with iterators, algorithms, and modern range-based code.

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

int main() {
    std::vector<int> values{5, 1, 4, 2, 3};

    std::sort(values.begin(), values.end());

    for (int value : values) {
        std::cout << value << ' ';
```

```
    }

    std::cout << '\n';
}
```

13.1.4 They embody well-understood trade-offs

Each standard container is built around a specific structure and performance model. Learning containers means learning how different data-organization strategies affect real programs.
That is one reason containers are such an important educational topic: they teach data-structure thinking through practical code.

13.2 `std::vector`

`std::vector` is the most important sequence container in Modern C++. It stores elements in contiguous memory and supports fast random access.
For many ordinary programming tasks, `std::vector` should be the first container you consider.

13.2.1 Basic usage

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

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

    values.push_back(40);
    values.push_back(50);

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

13.2.2 Why `std::vector` is so important

`std::vector` is often preferred because:

- it stores elements contiguously,
- it supports fast random access with `operator[]`,
- it grows dynamically,
- it works extremely well with algorithms,
- it is cache-friendly in many common scenarios.

13.2.3 Common operations

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

int main() {
    std::vector<int> values;

    values.push_back(1);
    values.push_back(2);
    values.push_back(3);

    std::cout << "Size: " << values.size() << '\n';
    std::cout << "First: " << values.front() << '\n';
    std::cout << "Last: " << values.back() << '\n';

    values.pop_back();

    for (std::size_t i{0}; i < values.size(); ++i) {
        std::cout << values[i] << ' ';
    }

    std::cout << '\n';
}
```

13.2.4 Insertion and resizing

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

int main() {
    std::vector<int> data(5, 100);

    data.resize(8, 200);

    for (int x : data) {
        std::cout << x << ' ';
    }

    std::cout << '\n';
}
```

13.2.5 Reserve and capacity

A vector may allocate more memory than its current size to support future growth efficiently.

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

int main() {
    std::vector<int> values;
    values.reserve(100);
```

```
    std::cout << "Size: " << values.size() << '\n';
    std::cout << "Capacity: " << values.capacity() << '\n';
}
```

13.2.6 When `std::vector` is a great choice

Use `std::vector` when:

- you need a resizable sequence,
- random access matters,
- you mostly add at the end,
- contiguous storage is beneficial,
- you want a general-purpose default sequence container.

13.2.7 A practical example

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

int main() {
    std::vector<int> scores{90, 85, 88, 92, 79};

    int total = std::accumulate(scores.begin(), scores.end(), 0);

    std::cout << "Average = "
              << static_cast<double>(total) / scores.size()
              << '\n';
}
```

13.3 `std::deque`

`std::deque` stands for double-ended queue. It is a sequence container that supports fast random access and efficient insertion and removal at both the front and the back.

13.3.1 Basic usage

```
#include <deque>
#include <iostream>

int main() {
    std::deque<int> values{20, 30, 40};

    values.push_front(10);
    values.push_back(50);
```

```
    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

13.3.2 Key characteristics

Compared with `std::vector`:

- `std::deque` also supports random access,
- it is efficient at both ends,
- it is not required to be stored as one single contiguous memory block.

13.3.3 Front and back operations

```
#include <deque>
#include <iostream>

int main() {
    std::deque<std::string> tasks;

    tasks.push_back("compile");
    tasks.push_back("link");
    tasks.push_front("configure");

    std::cout << "Front: " << tasks.front() << '\n';
    std::cout << "Back: " << tasks.back() << '\n';

    tasks.pop_front();
    tasks.pop_back();

    for (const auto& task : tasks) {
        std::cout << task << '\n';
    }
}
```

13.3.4 When `std::deque` is useful

Use `std::deque` when:

- you need fast insertion and removal at both ends,
- you still want random access,
- contiguous storage is not required,
- your data naturally grows from both sides.

13.3.5 A simple buffering example

```
#include <deque>
#include <iostream>

int main() {
    std::deque<int> recent_values;

    for (int i{1}; i <= 10; ++i) {
        recent_values.push_back(i);

        if (recent_values.size() > 5) {
            recent_values.pop_front();
        }
    }

    for (int value : recent_values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

13.4 `std::list`

`std::list` is a doubly linked-list container. It stores elements as nodes linked together rather than in contiguous storage.

13.4.1 Basic usage

```
#include <iostream>
#include <list>

int main() {
    std::list<int> values{10, 20, 30};

    values.push_front(5);
    values.push_back(40);

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

13.4.2 Key characteristics

`std::list`:

- supports efficient insertion and deletion anywhere once the position is known,

- does not support random access,
- uses more per-element overhead than contiguous sequence containers,
- is not cache-friendly compared with `std::vector`.

13.4.3 Insertion and erasure in the middle

```
#include <iostream>
#include <list>

int main() {
    std::list<int> values{1, 2, 4, 5};

    auto it = values.begin();
    std::advance(it, 2);

    values.insert(it, 3);

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

13.4.4 Why `std::list` is less common than many beginners expect

Beginners sometimes assume linked lists are naturally superior for insertion and deletion. In practice, `std::vector` is often faster in real workloads because contiguous storage and cache locality matter a great deal.
So `std::list` should not be chosen automatically just because insertion exists. Its trade-offs must match the real problem.

13.4.5 When `std::list` is appropriate

Use `std::list` when:

- stable node-based insertion and removal semantics matter,
- frequent mid-sequence insertion and erasure are central,
- random access is not needed,
- node-based structure truly matches the workload.

13.4.6 A small example of erasing while iterating

```
#include <iostream>
#include <list>

int main() {
    std::list<int> values{1, 2, 3, 4, 5, 6};
```

```
    for (auto it = values.begin(); it != values.end(); ) {
        if (*it % 2 == 0) {
            it = values.erase(it);
        } else {
            ++it;
        }
    }

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

13.5 `std::map`

`std::map` is an ordered associative container that stores key-value pairs with unique keys.

13.5.1 Basic usage

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

int main() {
    std::map<std::string, int> ages;

    ages["Ayman"] = 50;
    ages["Omar"] = 22;
    ages["Sara"] = 19;

    for (const auto& [name, age] : ages) {
        std::cout << name << " -> " << age << '\n';
    }
}
```

13.5.2 Key characteristics

`std::map`:

- stores elements ordered by key,
- requires unique keys,
- provides lookup by key,
- supports iteration in sorted key order.

13.5.3 Lookup and insertion

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

int main() {
    std::map<std::string, int> inventory{
        {"keyboard", 12},
        {"mouse", 20},
        {"monitor", 5}
    };

    auto it = inventory.find("mouse");

    if (it != inventory.end()) {
        std::cout << "mouse count = " << it->second << '\n';
    }

    inventory.insert({"laptop", 7});
}
```

13.5.4 Using `at()` versus `operator[]`

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

int main() {
    std::map<std::string, int> scores{
        {"Ali", 90},
        {"Nora", 95}
    };

    std::cout << scores["Ali"] << '\n';
    std::cout << scores.at("Nora") << '\n';
}
```

Important difference:

- `operator[]` may insert a default value if the key is missing,
- `at()` expects the key to exist and throws on missing access.

13.5.5 When `std::map` is a good choice

Use `std::map` when:

- key-value storage is needed,
- sorted order by key matters,
- unique keys are required,

- predictable ordered traversal is useful.

13.5.6 A frequency-count example

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

int main() {
    std::vector<std::string> words{
        "cpp", "rust", "cpp", "go", "cpp", "go"
    };

    std::map<std::string, int> counts;

    for (const auto& word : words) {
        ++counts[word];
    }

    for (const auto& [word, count] : counts) {
        std::cout << word << " : " << count << '\n';
    }
}
```

13.6 `std::unordered_map`

`std::unordered_map` is a hash-based associative container for key-value pairs with unique keys.

13.6.1 Basic usage

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

int main() {
    std::unordered_map<std::string, int> ages;

    ages["Ayman"] = 50;
    ages["Omar"] = 22;
    ages["Sara"] = 19;

    for (const auto& [name, age] : ages) {
        std::cout << name << " -> " << age << '\n';
    }
}
```

13.6.2 Key characteristics

`std::unordered_map`:

- is organized by hashing into buckets,
- does not maintain sorted key order,
- provides fast average-case lookup,
- uses unique keys.

13.6.3 Lookup example

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

int main() {
    std::unordered_map<std::string, int> ports{
        {"http", 80},
        {"https", 443},
        {"ssh", 22}
    };

    if (auto it = ports.find("https"); it != ports.end()) {
        std::cout << "https uses port " << it->second << '\n';
    }
}
```

13.6.4 When `std::unordered_map` is a good choice

Use `std::unordered_map` when:

- you need key-value lookup,
- sorted order is not required,
- fast average-case lookup is a priority,
- hashing is appropriate for the key type.

13.6.5 Bucket-related behavior

Because it is hash-based, its performance depends partly on hash quality and bucket distribution. That is one reason unordered containers are described in average-case terms rather than always-exact ordered-tree terms.

13.6.6 A counting example

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

int main() {
    std::vector<std::string> tokens{
```

```
        "if", "for", "if", "while", "for", "if"
    };

    std::unordered_map<std::string, int> freq;

    for (const auto& token : tokens) {
        ++freq[token];
    }

    for (const auto& [token, count] : freq) {
        std::cout << token << " -> " << count << '\n';
    }
}
```

13.7 `std::set`

`std::set` is an ordered associative container that stores unique keys only.

13.7.1 Basic usage

```
#include <iostream>
#include <set>

int main() {
    std::set<int> values{5, 2, 8, 2, 1, 5};

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

The duplicates are removed automatically, and iteration follows sorted order.

13.7.2 Key characteristics

`std::set`:

- stores only keys, not key-value pairs,
- keeps keys ordered,
- enforces uniqueness,
- is useful for ordered membership sets.

13.7.3 Insertion and membership testing

```
#include <iostream>
#include <set>
```

```
int main() {
    std::set<std::string> keywords;

    keywords.insert("class");
    keywords.insert("template");
    keywords.insert("namespace");
    keywords.insert("class");

    if (keywords.find("template") != keywords.end()) {
        std::cout << "Found template\n";
    }
}
```

13.7.4 When `std::set` is a good choice

Use `std::set` when:

- unique elements are required,
- sorted order matters,
- membership testing is needed,
- tree-ordered traversal is useful.

13.7.5 A practical example

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

int main() {
    std::vector<int> raw{4, 2, 7, 2, 9, 4, 1};
    std::set<int> unique_sorted(raw.begin(), raw.end());

    for (int value : unique_sorted) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

13.8 `std::unordered_set`

`std::unordered_set` is a hash-based associative container of unique keys.

13.8.1 Basic usage

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

```cpp
#include <unordered_set>

int main() {
    std::unordered_set<std::string> names{
        "Ayman", "Omar", "Sara", "Ayman"
    };

    for (const auto& name : names) {
        std::cout << name << '\n';
    }
}
```

13.8.2 Key characteristics

`std::unordered_set`:

- stores unique keys,
- uses hashing and buckets,
- does not preserve sorted order,
- is useful for fast average-case membership tests.

13.8.3 Membership checking

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

int main() {
    std::unordered_set<std::string> allowed{
        "read", "write", "execute"
    };

    if (allowed.find("write") != allowed.end()) {
        std::cout << "Permission exists\n";
    }
}
```

13.8.4 When `std::unordered_set` is a good choice

Use `std::unordered_set` when:

- uniqueness matters,
- sorted order does not matter,
- average-case fast membership lookup is more important than order.

13.8.5 A filtering example

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

int main() {
    std::vector<std::string> input{
        "cpp", "rust", "cpp", "go", "zig", "go"
    };

    std::unordered_set<std::string> seen;

    for (const auto& item : input) {
        if (seen.insert(item).second) {
            std::cout << "New item: " << item << '\n';
        }
    }
}
```

13.9 Choosing the right container

A container should be chosen according to the real needs of the program, not by habit.

13.9.1 A practical sequence-container guide

Use `std::vector` when:

- you need a general-purpose dynamic sequence,
- random access matters,
- appending at the end is common,
- contiguous storage is beneficial.

Use `std::deque` when:

- you need fast insertion and removal at both ends,
- random access still matters,
- strict contiguity is unnecessary.

Use `std::list` when:

- node-based insertion and erasure in arbitrary positions are central,
- random access is unimportant,
- list-specific semantics genuinely fit the workload.

13.9.2 A practical associative-container guide

Use `std::map` when:

- you need key-value storage,
- keys must be ordered,
- ordered traversal matters,
- uniqueness is required.

Use `std::unordered_map` when:

- key-value lookup speed is important,
- key order does not matter,
- a hash-based structure fits the use case.

Use `std::set` when:

- you need ordered unique keys,
- membership and sorted traversal matter.

Use `std::unordered_set` when:

- you need unique keys,
- order does not matter,
- fast average-case membership testing is the main goal.

13.9.3 A useful rule of thumb

For many beginners and many real-world programs:

- start with `std::vector` for sequences,
- start with `std::unordered_map` or `std::map` for key-value lookup depending on whether order matters,
- start with `std::unordered_set` or `std::set` for uniqueness depending on whether order matters.

Then refine the choice if profiling, interface needs, ordering semantics, or iterator guarantees suggest something different.

13.10 Basic complexity understanding

A serious programmer needs at least a practical understanding of complexity, even before studying formal algorithm analysis in depth.
Complexity describes how the cost of an operation grows as the amount of data grows.

13.10.1 Why complexity matters

If two containers both solve the same logical problem, their performance may still differ dramatically because the cost of insertion, lookup, traversal, or deletion is different.
That is why container choice is not only about syntax. It is also about cost models.

13.10.2 Simple informal terms

For beginners, the most useful intuition is:

- **constant time** means the operation cost does not grow much with container size,
- **linear time** means the operation cost grows roughly in proportion to the number of elements,
- **logarithmic time** means growth is much slower than linear and is typical of balanced tree lookup,
- **average constant time** in hash containers means lookups are usually very fast, but quality depends on hashing and distribution.

13.10.3 Sequence-container intuition

`std::vector`:

- random access is fast,
- push at the end is typically efficient,
- insertion or erasure in the middle may require moving later elements.

`std::deque`:

- random access is fast,
- insertion and deletion at both ends are efficient,
- middle insertions are less attractive than end operations.

`std::list`:

- no random access,
- insertion and erasure at known positions are efficient,
- traversal to reach a position is linear.

13.10.4 Associative-container intuition

`std::map` and `std::set`:

- ordered tree-based behavior,
- lookup, insertion, and erasure are typically logarithmic,

- iteration follows sorted key order.

`std::unordered_map` and `std::unordered_set`:

- hash-based behavior,
- lookup, insertion, and erasure are often average constant time,
- order is not meaningful in the sorted-tree sense.

13.10.5 A practical warning

Complexity is essential, but it is not the whole story.
Real performance also depends on:

- memory layout,
- cache locality,
- allocation behavior,
- key type cost,
- hash quality,
- constant factors,
- how the program really uses the container.

That is why `std::vector` often performs extremely well in practice, even when another container may appear theoretically attractive for a narrow operation.

13.10.6 A tiny intuition example

Searching in a sequence linearly:

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

int main() {
    std::vector<int> values{10, 20, 30, 40, 50};

    auto it = std::find(values.begin(), values.end(), 40);

    if (it != values.end()) {
        std::cout << "Found\n";
    }
}
```

Membership lookup in a hash set:

```cpp
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_set<int> values{10, 20, 30, 40, 50};

    if (values.find(40) != values.end()) {
        std::cout << "Found\n";
    }
}
```

These solve related problems, but they rely on very different container models.

Conclusion

Standard containers are one of the foundations of Modern C++ programming. They matter because they provide safe, reusable, expressive, and efficient data-structure abstractions that allow the programmer to focus on problem-solving rather than rebuilding storage logic manually.
The key sequence containers in this chapter each have distinct strengths:

- `std::vector` is the general-purpose dynamic sequence and the usual first choice,
- `std::deque` is useful when both ends matter,
- `std::list` is a node-based linked structure with specific insertion and erasure strengths.

The key associative containers also have clear roles:

- `std::map` stores ordered unique key-value pairs,
- `std::unordered_map` stores hashed unique key-value pairs,
- `std::set` stores ordered unique keys,
- `std::unordered_set` stores hashed unique keys.

Choosing the right container means matching the container's semantics and performance model to the actual needs of the program. That includes understanding whether:

- order matters,
- uniqueness matters,
- random access matters,
- insertion patterns matter,
- contiguous storage matters,
- average-case hashing behavior is acceptable.

A programmer who understands standard containers well is already thinking like a professional C++ developer, because container choice is one of the clearest ways that design, correctness, and performance meet in everyday code.

Iterators and Algorithms

Iterators and algorithms are among the most elegant and powerful parts of the C++ Standard Library. They embody one of the central design ideas of Modern C++: separate data structures from the operations performed on them.
Instead of building each operation directly into every container type, the Standard Library takes a more general approach. Containers provide access to their elements through iterators, and algorithms operate on iterator ranges. This design allows the same algorithm to work with many different containers, as long as the iterator requirements are satisfied.
For example, sorting, searching, transforming, counting, copying, and filtering do not have to be reimplemented separately for every sequence container. The algorithm works over a range, and the iterator abstracts how to traverse that range.
This chapter introduces:

- what iterators are,
- how `begin()` and `end()` define ranges,
- why `<algorithm>` is one of the most important Standard Library headers,
- how to use common algorithms such as `sort`, `find`, `count`, `transform`, `copy`, and `remove_if`,
- why separating data from algorithms leads to cleaner design,
- how algorithms help you write shorter, clearer, and more maintainable code.

14.1 Iterators

An iterator is an object that refers to elements in a sequence or container and allows traversal from one element to another. A useful way to think about iterators is that they generalize pointers.
A raw pointer can move through a built-in array and access elements. An iterator plays a similar role, but it works with many different Standard Library containers and container-like abstractions.

14.1.1 Why iterators exist

Without iterators, each container would need a completely separate interface for traversal, and each algorithm would need to be specialized for every data structure. Iterators solve that problem by providing a common traversal interface.
That means:

- algorithms can work with vectors, deques, lists, arrays, strings, and more,
- the algorithm does not need to know the exact container type,
- containers and algorithms remain decoupled.

14.1.2 A simple vector iterator example

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

int main() {
    std::vector<int> values{10, 20, 30, 40};

    std::vector<int>::iterator it = values.begin();

    std::cout << *it << '\n'; // 10

    ++it;
    std::cout << *it << '\n'; // 20
}
```

Here:

- `it` is an iterator,
- `*it` dereferences the iterator to access the current element,
- `++it` advances to the next element.

14.1.3 Iterating manually

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

int main() {
    std::vector<int> values{1, 2, 3, 4, 5};

    for (auto it = values.begin(); it != values.end(); ++it) {
        std::cout << *it << ' ';
    }

    std::cout << '\n';
}
```

This illustrates the classic iterator loop pattern:

- start at `begin()`,
- continue until `end()`,
- advance with `++it`,
- access the current element with `*it`.

14.1.4 Const iterators

When iteration should not modify the container elements, a const iterator is appropriate.

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

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

    for (auto it = values.cbegin(); it != values.cend(); ++it) {
        std::cout << *it << ' ';
    }

    std::cout << '\n';
}
```

`cbegin()` and `cend()` produce iterators that treat the elements as read-only through that iterator.

14.1.5 Iterators are not all equally powerful

Different containers provide different iterator capabilities. A vector iterator supports random access. A list iterator does not. This affects which algorithms can be used.
For example:

- `std::sort` requires random-access iterators,
- `std::find` can work with weaker iterator categories.

This is why not every algorithm works with every container.

14.1.6 Iterator invalidation matters

An important practical rule is that some container operations may invalidate iterators. For example, growing a vector may reallocate its storage, which can make previous iterators invalid.
Example of dangerous thinking:

```cpp
#include <vector>

int main() {
    std::vector<int> values{1, 2, 3};

    auto it = values.begin();
    values.push_back(4);

    // it may now be invalid depending on reallocation
}
```

This chapter focuses on algorithmic usage, but the broader lesson is important: iterators are powerful, but they are only valid as long as the container guarantees their validity.

14.2 `begin/end`

The pair `begin()` and `end()` defines a half-open range:

- `begin()` refers to the first element,
- `end()` refers to the position one past the last element.

This design is one of the most important patterns in the Standard Library.

14.2.1 Why half-open ranges are useful

A half-open range `[first, last)` has several advantages:

- it can represent an empty range naturally when `first == last`,
- the length can often be computed as the difference between endpoints,
- adjacent ranges fit together cleanly,
- iteration stops before dereferencing `end()`.

14.2.2 Basic example

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

int main() {
    std::vector<int> values{100, 200, 300};

    auto first = values.begin();
    auto last = values.end();

    for (auto it = first; it != last; ++it) {
        std::cout << *it << '\n';
    }
}
```

14.2.3 Important rule: never dereference `end()`

`end()` is a sentinel-like position past the last valid element. It is used for comparisons, not for direct dereferencing.
Incorrect:

```
// auto it = values.end();
// std::cout << *it; // invalid
```

14.2.4 Global `std::begin` and `std::end`

In addition to member functions such as `values.begin()`, there are also global helper functions `std::begin(values)` and `std::end(values)`.

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

int main() {
```

```
    std::array<int, 4> data{1, 2, 3, 4};

    for (auto it = std::begin(data); it != std::end(data); ++it) {
        std::cout << *it << ' ';
    }

    std::cout << '\n';
}
```

This form is especially useful because it also works naturally with built-in arrays.

14.2.5 Built-in array example

```
#include <iostream>
#include <iterator>

int main() {
    int data[]{5, 10, 15, 20};

    for (auto it = std::begin(data); it != std::end(data); ++it) {
        std::cout << *it << ' ';
    }

    std::cout << '\n';
}
```

That is one of the reasons the global `begin`/`end` functions are so useful: they help unify array-style and container-style code.

14.2.6 Const range endpoints

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

int main() {
    std::vector<int> values{2, 4, 6, 8};

    for (auto it = std::cbegin(values); it != std::cend(values); ++it) {
        std::cout << *it << ' ';
    }

    std::cout << '\n';
}
```

14.3 <algorithm>

The header <algorithm> defines many of the Standard Library's most important algorithms. These algorithms operate on ranges through iterators.

This is one of the most important ideas in Modern C++:

- containers manage storage,
- algorithms express operations.

14.3.1 Why <algorithm> matters

The algorithms in <algorithm> matter because they:

- reduce hand-written loop boilerplate,
- encourage well-tested library solutions,
- make code more expressive,
- allow the same operation to work across different data structures.

14.3.2 General range form

Most classic algorithms work with iterator pairs:

```
algorithm_name(first, last, ...);
```

This means:

- operate on the half-open range [first, last),
- do not care which container produced that range,
- rely on iterator capabilities rather than container-specific APIs.

14.3.3 An early simple example

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

int main() {
    std::vector<int> values{3, 1, 4, 1, 5};

    auto it = std::find(values.begin(), values.end(), 4);

    if (it != values.end()) {
        std::cout << "Found: " << *it << '\n';
    }
}
```

14.3.4 Algorithms are not limited to one container type

Because algorithms use iterators, the same algorithm may work with:

- std::vector,
- std::deque,
- std::list,
- std::array,

- built-in arrays,
- and many user-defined ranges that provide the required iterator behavior.

That separation is one of the key design successes of the Standard Library.

14.4 `sort`, `find`, `count`, `transform`, `copy`, `remove_if`

This section introduces some of the most important and frequently used Standard Library algorithms.

14.4.1 `sort`

`std::sort` orders elements in a range. It requires random-access iterators, which means it works with containers such as `std::vector`, `std::array`, and `std::deque`, but not with `std::list`.

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

int main() {
    std::vector<int> values{5, 2, 8, 1, 4};

    std::sort(values.begin(), values.end());

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

14.4.2 Custom comparison

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

int main() {
    std::vector<int> values{5, 2, 8, 1, 4};

    std::sort(values.begin(), values.end(),
              [](int a, int b) {
                  return a > b;
              });

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

14.4.3 `find`

`std::find` searches for the first occurrence of a value in a range.

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

int main() {
    std::vector<int> values{10, 20, 30, 40, 50};

    auto it = std::find(values.begin(), values.end(), 30);

    if (it != values.end()) {
        std::cout << "Found value: " << *it << '\n';
    } else {
        std::cout << "Value not found\n";
    }
}
```

14.4.4 `count`

`std::count` counts how many elements in a range are equal to a given value.

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

int main() {
    std::vector<int> values{1, 2, 2, 3, 2, 4};

    auto result = std::count(values.begin(), values.end(), 2);

    std::cout << "Number of 2s: " << result << '\n';
}
```

14.4.5 `transform`

`std::transform` applies an operation to elements and writes the transformed results to an output range.

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

int main() {
    std::vector<int> values{1, 2, 3, 4};
    std::vector<int> doubled(values.size());

    std::transform(values.begin(), values.end(),
                   doubled.begin(),
                   [](int x) {
                       return x * 2;
                   });
```

```
    for (int value : doubled) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

14.4.6 In-place transform

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

int main() {
    std::vector<int> values{1, 2, 3, 4};

    std::transform(values.begin(), values.end(),
                   values.begin(),
                   [](int x) {
                       return x * x;
                   });

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

14.4.7 copy

`std::copy` copies elements from one range into another output range.

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

int main() {
    std::vector<int> source{10, 20, 30, 40};
    std::vector<int> destination(source.size());

    std::copy(source.begin(), source.end(), destination.begin());

    for (int value : destination) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

14.4.8 Copying into a growing destination

Sometimes the destination should grow automatically. In that case, use an inserter such as `std::back_inserter`.

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main() {
    std::vector<int> source{1, 2, 3, 4};
    std::vector<int> destination;

    std::copy(source.begin(), source.end(),
              std::back_inserter(destination));

    for (int value : destination) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

14.4.9 `remove_if`

`std::remove_if` is one of the most important algorithms to understand correctly. It does not directly erase elements from a container. Instead, it reorders the range so that the elements to keep are moved to the front, and it returns a new logical end.

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

int main() {
    std::vector<int> values{1, 2, 3, 4, 5, 6};

    auto new_end = std::remove_if(values.begin(), values.end(),
                                  [](int x) {
                                      return x % 2 == 0;
                                  });

    for (auto it = values.begin(); it != new_end; ++it) {
        std::cout << *it << ' ';
    }

    std::cout << '\n';
}
```

14.4.10 The erase-remove idiom

To actually remove the unwanted elements from a container such as `std::vector`, combine `remove_if` with `erase`.

```
#include <algorithm>
```

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

int main() {
    std::vector<int> values{1, 2, 3, 4, 5, 6};

    values.erase(
        std::remove_if(values.begin(), values.end(),
                       [](int x) {
                           return x % 2 == 0;
                       }),
        values.end()
    );

    for (int value : values) {
        std::cout << value << ' ';
    }

    std::cout << '\n';
}
```

This is known as the erase-remove idiom and is an essential Modern C++ pattern for sequence containers.

14.4.11 A combined example

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main() {
    std::vector<int> values{5, 1, 4, 2, 3, 2};

    std::sort(values.begin(), values.end());

    std::cout << "Sorted: ";
    for (int value : values) {
        std::cout << value << ' ';
    }
    std::cout << '\n';

    std::cout << "Count of 2: "
              << std::count(values.begin(), values.end(), 2)
              << '\n';

    auto it = std::find(values.begin(), values.end(), 4);
    if (it != values.end()) {
        std::cout << "Found 4\n";
    }

    std::vector<int> squares;
    std::transform(values.begin(), values.end(),
```

```
                    std::back_inserter(squares),
                    [](int x) {
                        return x * x;
                    });

    std::cout << "Squares: ";
    for (int value : squares) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

14.5 Separating data from algorithms

One of the deepest and most important ideas in the Standard Library is the separation between:

- the structure that stores data,
- the operation that processes it.

14.5.1 Why this separation is powerful

If algorithms were built directly into each container as custom member functions, code would be more repetitive and less general. Instead, the Standard Library says:

> Represent traversal through iterators, then let algorithms work on ranges.

This produces more reusable and composable code.

14.5.2 A container-specific mindset versus a range-based mindset

A beginner might think:

> "I have a vector, so I need vector-specific search code."

Modern C++ encourages:

> "I have a range of elements. I should use a general algorithm over that range."

That shift in thinking is very important.

14.5.3 Example: searching manually

Manual loop:

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

int main() {
    std::vector<int> values{10, 20, 30, 40};
    bool found{false};
```

```
    for (int value : values) {
        if (value == 30) {
            found = true;
            break;
        }
    }

    std::cout << (found ? "Found\n" : "Not found\n");
}
```

Algorithmic style:

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

int main() {
    std::vector<int> values{10, 20, 30, 40};

    bool found = std::find(values.begin(), values.end(), 30) != values.end();

    std::cout << (found ? "Found\n" : "Not found\n");
}
```

The second version expresses intent more directly.

14.5.4 Same algorithm, different containers

Because algorithms are decoupled from containers, a similar pattern works for different structures.
With a vector:

```
std::find(vec.begin(), vec.end(), 42);
```

With a deque:

```
std::find(dq.begin(), dq.end(), 42);
```

With a built-in array:

```
std::find(std::begin(arr), std::end(arr), 42);
```

The algorithm remains the same. Only the range changes.

14.5.5 Cleaner interfaces through ranges

Once you think in terms of iterator ranges, you can write functions that operate on generic ranges rather than on one exact concrete container type.
Example:

```cpp
#include <algorithm>
#include <iostream>

template <typename Iterator>
void print_sorted(Iterator first, Iterator last) {
    std::sort(first, last);

    for (auto it = first; it != last; ++it) {
        std::cout << *it << ' ';
    }

    std::cout << '\n';
}
```

This is much more general than writing a function that only accepts one specific container type.

14.6 Writing cleaner and shorter code

One of the practical rewards of learning algorithms well is that code becomes shorter, clearer, and more declarative.

14.6.1 From loops to intent

Many loops in beginner code exist only because the programmer has not yet learned the algorithm that expresses the same intent directly.
For example, manual counting:

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

int main() {
    std::vector<int> values{1, 2, 2, 3, 2};
    int count{0};

    for (int value : values) {
        if (value == 2) {
            ++count;
        }
    }

    std::cout << count << '\n';
}
```

Cleaner algorithmic version:

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> values{1, 2, 2, 3, 2};
```

```
    std::cout << std::count(values.begin(), values.end(), 2) << '\n';
}
```

14.6.2 Transforming data clearly

Manual transformation:

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

int main() {
    std::vector<int> input{1, 2, 3, 4};
    std::vector<int> output;

    for (int x : input) {
        output.push_back(x * 10);
    }

    for (int x : output) {
        std::cout << x << ' ';
    }

    std::cout << '\n';
}
```

Algorithmic transformation:

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main() {
    std::vector<int> input{1, 2, 3, 4};
    std::vector<int> output;

    std::transform(input.begin(), input.end(),
                   std::back_inserter(output),
                   [](int x) {
                       return x * 10;
                   });

    for (int x : output) {
        std::cout << x << ' ';
    }

    std::cout << '\n';
}
```

14.6.3 Removing elements clearly

Manual removal code is often error-prone. The erase-remove idiom is shorter and more standard.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> values{1, 2, 3, 4, 5, 6};

    values.erase(
        std::remove_if(values.begin(), values.end(),
                       [](int x) {
                           return x < 4;
                       }),
        values.end()
    );

    for (int x : values) {
        std::cout << x << ' ';
    }

    std::cout << '\n';
}
```

14.6.4 Why shorter code is not the only goal

The goal is not merely to write fewer lines. The real goal is to write code whose intent is immediately visible.
Compare:

- a hand-written loop that must be read carefully to infer what it does,
- an algorithm call whose name already states the intended operation.

That is why algorithms often improve readability, not just brevity.

14.6.5 A practical style guideline

Prefer a Standard Library algorithm when:

- the algorithm clearly expresses the task,
- the needed operation already exists,
- the result is clearer than a manual loop.

Write a manual loop when:

- the logic is genuinely custom,
- forcing a standard algorithm would reduce clarity,
- multiple interdependent steps are easier to understand in explicit code.

14.6.6 An end-to-end example

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

int main() {
    std::vector<int> data{9, 3, 7, 1, 8, 2, 6, 4, 5};

    std::sort(data.begin(), data.end());

    data.erase(
        std::remove_if(data.begin(), data.end(),
                       [](int x) {
                           return x % 2 == 0;
                       }),
        data.end()
    );

    std::vector<int> squared;
    std::transform(data.begin(), data.end(),
                   std::back_inserter(squared),
                   [](int x) {
                       return x * x;
                   });

    std::cout << "Result: ";
    std::copy(squared.begin(), squared.end(),
              std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';
}
```

This compact example shows the Standard Library style clearly:

- sort the data,
- remove even values,
- transform remaining values,
- copy results to output.

The program reads almost like a direct description of the task.

Conclusion

Iterators and algorithms are central to the design philosophy of Modern C++. Iterators provide a uniform way to traverse data structures, and algorithms operate on iterator ranges rather than on specific container types.
This separation is powerful because it makes code:

- more reusable,

- more generic,
- more expressive,
- often shorter and clearer.

The pair `begin()` and `end()` defines the fundamental half-open range model used throughout the Standard Library. The header `<algorithm>` provides a broad collection of operations that work over those ranges.
The algorithms introduced in this chapter are especially important:

- `sort` for ordering,
- `find` for searching,
- `count` for counting matches,
- `transform` for element-wise conversion,
- `copy` for range copying,
- `remove_if` for logical filtering, often combined with `erase`.

Learning to use these algorithms well is a major step toward writing professional C++ code. It shifts the programmer away from repetitive hand-written loops and toward clearer statements of intent. That shift is one of the marks of Modern C++ style.

Part V

Object-Oriented Programming in Modern C++

Classes and Objects

Classes are one of the central building blocks of C++. They allow programmers to define their own types, combining data and behavior into a single coherent unit. This ability to model objects with state and operations is one of the foundations of object-oriented programming, but it is also more general than that. In Modern C++, classes are used not only for traditional object-oriented design, but also for resource management, value types, generic components, domain modeling, and many forms of abstraction.
A class allows a programmer to answer several essential design questions:

- What data belongs together?
- What operations are valid on that data?
- Which parts should be visible to users of the type?
- Which parts should remain internal implementation details?

This chapter introduces the fundamental structure of classes and objects in Modern C++. It explains:

- what a class is,
- how member variables and member functions work,
- how `public` and `private` control access,
- how objects are created,
- why separating interface from implementation leads to better design.

The goal is not merely to show syntax. The goal is to build the right mental model for designing and using types correctly in Modern C++.

15.1 What is a class

A class is a user-defined type that groups related data and functions together. The data describes the state of an object, and the functions describe the operations that can be performed on that object.
A simple class definition looks like this:

```
class Counter {
    int value_;
};
```

This defines a new type named `Counter`. The class contains one data member named `value_`.
A slightly richer example includes behavior:

```
class Counter {
public:
    void increment() {
        ++value_;
    }

    int get() const {
        return value_;
    }

private:
    int value_{0};
};
```

Now the class represents both:

- state, through `value_`,
- behavior, through `increment()` and `get()`.

15.1.1 A class defines a type, not an object

A class definition creates a new type. An object is created later from that type.

```
class Point {
public:
    int x{0};
    int y{0};
};

int main() {
    Point p;
}
```

Here:

- `Point` is the class type,
- `p` is an object of type `Point`.

15.1.2 Class versus struct

In C++, `class` and `struct` are very similar. The key default difference is:

- members of a `class` are `private` by default,
- members of a `struct` are `public` by default.

Example with a class:

```
class Example {
    int value_;
};
```

Here `value_` is private by default.
Example with a struct:

```
struct Example {
    int value_;
};
```

Here `value_` is public by default.
For this chapter, the focus remains on `class`, because it is the standard form when teaching encapsulation and interface design.

15.1.3 Why classes matter

Classes matter because they help organize programs around meaningful abstractions. Instead of scattering related data and helper functions across a program, a class packages them together.
Without a class, code may become fragmented:

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

std::string person_name;
int person_age;

void print_person() {
    std::cout << person_name << " is " << person_age << " years old.\n";
}
```

This is weak design because the data and its operations are not grouped into a proper type.
A class-based version is clearer:

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

class Person {
public:
    void print() const {
        std::cout << name_ << " is " << age_ << " years old.\n";
    }

    void set_name(const std::string& name) {
        name_ = name;
    }

    void set_age(int age) {
        age_ = age;
    }

private:
    std::string name_;
```

```
    int age_{0};
};
```

This is easier to understand, extend, and maintain.

15.2 Member variables and functions

A class consists of members. The two most important member categories are:

- data members, also called member variables,
- member functions, also called methods.

15.2.1 Member variables

Member variables store the state of an object.

```
class Rectangle {
private:
    int width_{0};
    int height_{0};
};
```

Each object of type `Rectangle` has its own `width_` and `height_` values.

15.2.2 Member functions

Member functions operate on the object's state.

```
class Rectangle {
public:
    void set_width(int width) {
        width_ = width;
    }

    void set_height(int height) {
        height_ = height;
    }

    int area() const {
        return width_ * height_;
    }

private:
    int width_{0};
    int height_{0};
};
```

This class has:

- two data members: `width_` and `height_`,
- three member functions: `set_width()`, `set_height()`, and `area()`.

15.2.3 Using member functions

```
#include <iostream>

class Rectangle {
public:
    void set_width(int width) {
        width_ = width;
    }

    void set_height(int height) {
        height_ = height;
    }

    int area() const {
        return width_ * height_;
    }

private:
    int width_{0};
    int height_{0};
};

int main() {
    Rectangle r;
    r.set_width(10);
    r.set_height(5);

    std::cout << r.area() << '\n';
}
```

15.2.4 The implicit object parameter

Inside a non-static member function, the function operates on a particular object. The language provides this through the implicit `this` mechanism.
For example:

```
class Counter {
public:
    void increment() {
        ++value_;
    }

private:
    int value_{0};
};
```

When `increment()` is called on an object, it modifies that object's `value_`.

15.2.5 Const member functions

A member function may be marked `const` to indicate that it does not modify the observable state of the object.

```
class Counter {
public:
    int get() const {
        return value_;
    }

private:
    int value_{0};
};
```

The `const` after the parameter list is very important. It allows the function to be called on const objects and communicates that the function is read-only.

15.2.6 A complete example with members

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

class Book {
public:
    void set_title(const std::string& title) {
        title_ = title;
    }

    void set_pages(int pages) {
        pages_ = pages;
    }

    const std::string& title() const {
        return title_;
    }

    int pages() const {
        return pages_;
    }

    void print() const {
        std::cout << "Title: " << title_
                  << ", Pages: " << pages_
                  << '\n';
    }

private:
    std::string title_;
    int pages_{0};
};

int main() {
    Book book;
    book.set_title("Modern C++");
    book.set_pages(600);
    book.print();
```

```
}
```

This example shows a typical class layout:

- private state,
- public functions to modify and inspect that state,
- a clean and readable interface.

15.3 `public` / `private`

Access control is one of the most important ideas in class design. It allows a class to expose some members to users while hiding internal details.
The main access specifiers are:

- `public`
- `private`

This chapter focuses on these two because they are the most fundamental for beginner and intermediate class design.

15.3.1 `public`

Members declared under `public:` can be accessed from outside the class.

```
class Counter {
public:
    void increment() {
        ++value_;
    }

    int get() const {
        return value_;
    }

private:
    int value_{0};
};
```

Here:

- `increment()` is public,
- `get()` is public,
- both can be called by users of the class.

15.3.2 `private`

Members declared under `private:` can be accessed only from within the class itself and from selected friends, not from ordinary outside code.

```
class Counter {
public:
    void increment() {
        ++value_;
    }

private:
    int value_{0};
};
```

Outside code cannot access `value_` directly:

```
int main() {
    Counter c;
    // c.value_ = 10; // error
}
```

15.3.3 Why access control matters

Access control allows a class to separate:

- what users are allowed to do,
- how the class actually stores and manages its internal state.

This is essential for encapsulation.
Without `private`, any user of the class could modify internal state directly, making it much harder to enforce invariants or change implementation later.

15.3.4 A weak design with public data

```
class Temperature {
public:
    double celsius_;
};
```

This design exposes the internal representation directly. Any code may write arbitrary values without restriction.

15.3.5 A better encapsulated design

```
class Temperature {
public:
    void set_celsius(double value) {
        celsius_ = value;
    }

    double celsius() const {
```

```
        return celsius_;
    }

private:
    double celsius_{0.0};
};
```

Now the class controls access to its state through functions.

15.3.6 Access specifiers apply to following members

An access specifier affects all subsequent members until another access specifier appears.

```
class Sample {
public:
    void f1() {}
    void f2() {}

private:
    int x_{0};
    int y_{0};

public:
    int sum() const {
        return x_ + y_;
    }
};
```

15.3.7 A practical design rule

A strong default rule in Modern C++ is:

- keep data members private,
- expose behavior through public member functions,
- reveal only what the user of the type actually needs.

This does not mean every class must be complicated. It means access should be intentional.

15.4 Object creation

A class defines a type. An object is an instance of that type.

15.4.1 Creating an object on automatic storage

The most common form of object creation is ordinary local object definition.

```
class Point {
public:
    int x{0};
```

```
    int y{0};
};

int main() {
    Point p;
}
```

Here p is an object of type `Point`.

15.4.2 Using the object

```
#include <iostream>

class Point {
public:
    int x{0};
    int y{0};
};

int main() {
    Point p;
    p.x = 10;
    p.y = 20;

    std::cout << p.x << ", " << p.y << '\n';
}
```

15.4.3 Object creation with encapsulated classes

For a class with private members, creation still works the same way, but interaction occurs through public functions.

```
#include <iostream>

class Counter {
public:
    void increment() {
        ++value_;
    }

    int get() const {
        return value_;
    }

private:
    int value_{0};
};

int main() {
    Counter c;
    c.increment();
    c.increment();
```

```
    std::cout << c.get() << '\n';
}
```

15.4.4 Each object has its own state

Different objects of the same class hold their own separate member values.

```
#include <iostream>

class Counter {
public:
    void increment() {
        ++value_;
    }

    int get() const {
        return value_;
    }

private:
    int value_{0};
};

int main() {
    Counter a;
    Counter b;

    a.increment();
    a.increment();
    b.increment();

    std::cout << a.get() << '\n'; // 2
    std::cout << b.get() << '\n'; // 1
}
```

15.4.5 Aggregate-like versus encapsulated creation

Some simple types may expose public data directly, but many well-designed classes prefer encapsulated state.
Simple public-member object:

```
struct Position {
    int row{0};
    int col{0};
};

int main() {
    Position p{3, 5};
}
```

Encapsulated class:

```
class BankAccount {
```

```
public:
    void deposit(double amount) {
        balance_ += amount;
    }

    double balance() const {
        return balance_;
    }

private:
    double balance_{0.0};
};

int main() {
    BankAccount account;
    account.deposit(100.0);
}
```

Both define objects, but the second places stronger control around valid operations.

15.4.6 Dynamic object creation exists but is not the starting point

Objects can also be created dynamically, but that is not the best starting point for learning ordinary class use.
Example:

```
class Item {
public:
    int value{0};
};

int main() {
    Item* p = new Item{};
    delete p;
}
```

Modern C++ generally prefers ordinary automatic objects and RAII-based ownership abstractions rather than starting with raw `new` and `delete`. Later chapters will cover this in depth.

15.5 Interface vs implementation

One of the most important design ideas in class-based programming is the distinction between interface and implementation.

15.5.1 What is the interface?

The interface is the set of operations that users of the class are allowed to see and call. In most simple cases, that means the public member functions and any public types or constants intentionally exposed by the class.
Example interface:

```
class Counter {
public:
```

```
    void increment();
    int get() const;
};
```

This tells a user:

- the class can be incremented,
- the current value can be read.

15.5.2 What is the implementation?

The implementation is how the class internally stores its state and performs its work.
Example:

```
class Counter {
public:
    void increment() {
        ++value_;
    }

    int get() const {
        return value_;
    }

private:
    int value_{0};
};
```

Here:

- `increment()` and `get()` form the visible interface,
- `value_` is part of the implementation detail.

15.5.3 Why the distinction matters

Separating interface from implementation matters because it allows the internal design of the class to change without breaking user code, as long as the public interface remains stable.
Suppose a counter was first implemented with an `int`:

```
class Counter {
public:
    void increment() {
        ++value_;
    }

    int get() const {
        return value_;
    }

private:
    int value_{0};
};
```

Later, the implementation might change:

```
class Counter {
public:
    void increment() {
        value_ += step_;
    }

    int get() const {
        return value_;
    }

private:
    int value_{0};
    int step_{1};
};
```

The interface is unchanged, so calling code can remain the same.

15.5.4 A poor design that exposes implementation

```
class Counter {
public:
    int value_{0};
};
```

Now users depend directly on the data layout and internal representation. Changing the implementation becomes harder because outside code may rely on details that should never have been exposed.

15.5.5 A cleaner header/source separation

In real projects, interface and implementation are often separated physically into header and source files.
Header file:

```
// counter.hpp
#pragma once

class Counter {
public:
    void increment();
    int get() const;

private:
    int value_{0};
};
```

Source file:

```
// counter.cpp
#include "counter.hpp"

void Counter::increment() {
```

```
    ++value_;
}

int Counter::get() const {
    return value_;
}
```

User code:

```
// main.cpp
#include <iostream>
#include "counter.hpp"

int main() {
    Counter c;
    c.increment();
    c.increment();

    std::cout << c.get() << '\n';
}
```

This is a standard and important pattern:

- the header presents the type and its public interface,
- the source file provides the implementation.

15.5.6 Benefits of interface/implementation separation

This separation improves design because it:

- reduces coupling,
- hides details,
- makes code easier to maintain,
- allows internal changes with fewer external effects,
- makes the class easier to understand as a public abstraction.

15.5.7 A practical design example

Consider a simple bank account.
Bad design:

```
class BankAccount {
public:
    double balance_{0.0};
};
```

Any code can write invalid values directly.
Better design:

```cpp
#include <iostream>

class BankAccount {
public:
    void deposit(double amount) {
        if (amount > 0.0) {
            balance_ += amount;
        }
    }

    bool withdraw(double amount) {
        if (amount <= 0.0 || amount > balance_) {
            return false;
        }

        balance_ -= amount;
        return true;
    }

    double balance() const {
        return balance_;
    }

private:
    double balance_{0.0};
};

int main() {
    BankAccount account;
    account.deposit(200.0);

    if (account.withdraw(50.0)) {
        std::cout << "Remaining balance: " << account.balance() << '\n';
    }
}
```

This design makes the valid operations explicit and protects the state from arbitrary misuse.

Conclusion

Classes and objects are fundamental to Modern C++ because they allow programmers to define their own types and express meaningful abstractions.
A class:

- groups related state and behavior,
- defines member variables and member functions,
- controls access through `public` and `private`,
- serves as a blueprint from which objects are created.

The most important design lesson of this chapter is not merely that classes exist, but that good class design depends on encapsulation and clear interfaces.
A strong Modern C++ class usually follows these principles:

- keep implementation details private,
- expose only the operations users truly need,
- make object state and behavior consistent,
- separate interface from implementation so the type remains maintainable over time.

Once these ideas are understood, the programmer is ready to move deeper into class construction, initialization, special member functions, RAII, and the broader design techniques that make classes truly powerful in Modern C++.

Constructors and Destructors

Constructors and destructors are among the most important mechanisms in C++. They define how objects begin their life and how they end it. Without understanding them properly, it is impossible to understand initialization, resource management, object invariants, and many of the core design principles of Modern C++.
A constructor establishes the initial valid state of an object. A destructor performs cleanup when the object is destroyed. Together, they form the basis of RAII, one of the central ideas of safe C++ programming: acquire resources during initialization, and release them automatically during destruction.
Modern C++ also gives fine control over special member functions:

- some may be generated automatically by the compiler,
- some may be explicitly requested with `= default`,
- some may be forbidden with `= delete`.

This chapter explains:

- what constructors are and how they work,
- what a destructor does,
- how defaulted and deleted functions express intent,
- why member initializer lists are essential,
- what a default constructor is,
- the basic meaning of copy and move operations.

The purpose is not only to learn syntax, but to understand object life-cycle design in Modern C++.

16.1 Constructors

A constructor is a special member function that is used to initialize objects of a class. It has the same name as the class and has no return type.
A basic constructor example:

```
#include <iostream>

class Counter {
```

```
public:
    Counter() {
        std::cout << "Counter constructed\n";
    }
};

int main() {
    Counter c;
}
```

When c is created, the constructor is called automatically.

16.1.1 Why constructors matter

A constructor is responsible for establishing a valid initial state. This is one of the most important design duties of a class. Without a constructor, a type may leave important members uninitialized or inconsistent. With a constructor, a class can guarantee that every object starts life in a meaningful state.
Example:

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

class Person {
public:
    Person() {
        name_ = "Unknown";
        age_ = 0;
    }

    void print() const {
        std::cout << name_ << " : " << age_ << '\n';
    }

private:
    std::string name_;
    int age_;
};

int main() {
    Person p;
    p.print();
}
```

16.1.2 Constructors can take parameters

A constructor may accept arguments to initialize the object with caller-provided values.

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

class Person {
```

```
public:
    Person(const std::string& name, int age) {
        name_ = name;
        age_ = age;
    }

    void print() const {
        std::cout << name_ << " : " << age_ << '\n';
    }

private:
    std::string name_;
    int age_;
};

int main() {
    Person p("Ayman", 50);
    p.print();
}
```

16.1.3 Constructor overloading

A class may have more than one constructor, as long as their parameter lists differ.

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

class Person {
public:
    Person() {
        name_ = "Unknown";
        age_ = 0;
    }

    Person(const std::string& name, int age) {
        name_ = name;
        age_ = age;
    }

    void print() const {
        std::cout << name_ << " : " << age_ << '\n';
    }

private:
    std::string name_;
    int age_;
};

int main() {
    Person a;
    Person b("Sara", 25);
```

```
    a.print();
    b.print();
}
```

16.1.4 A constructor has no ordinary return type

This is invalid:

```
class Example {
public:
    // int Example() { return 0; } // invalid
};
```

A constructor does not return a value. Its role is to initialize the object being created.

16.1.5 Constructors run automatically

Constructors are not called like ordinary member functions after object creation. They are part of object creation itself.

```
class Number {
public:
    Number(int x) : value_{x} {}

private:
    int value_;
};

int main() {
    Number n(42);
}
```

The constructor runs during the creation of `n`.

16.1.6 Constructors and class invariants

A constructor should establish the invariants of the class. An invariant is a property that should always hold for valid objects. For example, if a class represents a non-negative balance, the constructor should ensure that the balance starts valid.

```
class BankAccount {
public:
    BankAccount(double initial_balance) {
        if (initial_balance < 0.0) {
            balance_ = 0.0;
        } else {
            balance_ = initial_balance;
        }
    }

    double balance() const {
        return balance_;
    }
```

```
private:
    double balance_{0.0};
};
```

16.2 Destructor

A destructor is a special member function that is called automatically when an object is destroyed. Its name is the class name prefixed with ~.

```
#include <iostream>

class Logger {
public:
    Logger() {
        std::cout << "Logger constructed\n";
    }

    ~Logger() {
        std::cout << "Logger destroyed\n";
    }
};

int main() {
    Logger log;
}
```

When `log` goes out of scope, the destructor runs automatically.

16.2.1 Why destructors matter

Destructors are used for cleanup. In Modern C++, that usually means releasing resources owned by the object, such as:

- files,
- memory,
- locks,
- sockets,
- operating system handles.

This is the heart of RAII.

16.2.2 A file-style example

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

class FileHandle {
public:
```

```
    explicit FileHandle(const std::string& name) : name_{name} {
        std::cout << "Opening " << name_ << '\n';
    }

    ~FileHandle() {
        std::cout << "Closing " << name_ << '\n';
    }

private:
    std::string name_;
};

int main() {
    FileHandle file("data.txt");
}
```

This demonstrates the idea clearly: construction acquires, destruction releases.

16.2.3 Destructors take no parameters

A destructor cannot have parameters and cannot be overloaded.

```
class Example {
public:
    ~Example() {}
};
```

There is only one destructor per class.

16.2.4 Automatic destruction order

For local objects, destruction happens automatically when scope ends, in reverse order of construction.

```
#include <iostream>

class Trace {
public:
    explicit Trace(const char* name) : name_{name} {
        std::cout << "Construct " << name_ << '\n';
    }

    ~Trace() {
        std::cout << "Destroy " << name_ << '\n';
    }

private:
    const char* name_;
};

int main() {
    Trace a("A");
    Trace b("B");
}
```

Conceptually, this produces:

```
Construct A
Construct B
Destroy B
Destroy A
```

16.2.5 Destructors and dynamic objects

If an object is created with `new`, its destructor runs when `delete` is used.

```cpp
#include <iostream>

class Item {
public:
    ~Item() {
        std::cout << "Item destroyed\n";
    }
};

int main() {
    Item* p = new Item{};
    delete p;
}
```

Modern C++ prefers avoiding raw owning pointers and manual `delete` in normal code, but the destructor rule remains the same.

16.3 Defaulted / deleted functions

Modern C++ allows explicit control over compiler-generated special member functions through:

- `= default`
- `= delete`

This is one of the most important modern language improvements for class design.

16.3.1 `= default`

Using `= default` tells the compiler to generate the usual default implementation for a special member function.
Example:

```cpp
class Widget {
public:
    Widget() = default;
};
```

This explicitly requests the compiler-generated default constructor.

16.3.2 Why use `= default`

A function may be defaulted to:

- express intent clearly,
- preserve compiler-generated behavior,
- make the class interface explicit,
- request generation even when writing the function declaration manually.

Example with destructor:

```
class Widget {
public:
    ~Widget() = default;
};
```

16.3.3 `= delete`

Using `= delete` explicitly forbids a function from being used.
Example:

```
class NonCopyable {
public:
    NonCopyable() = default;
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;
};
```

Now objects of `NonCopyable` cannot be copied.

16.3.4 Why use `= delete`

Deleted functions are useful when:

- an operation would be unsafe,
- copying should not be allowed,
- a conversion should be forbidden,
- ownership or identity semantics must be protected.

16.3.5 Example: a unique resource holder

```
#include <iostream>

class UniqueResource {
public:
    UniqueResource() = default;
```

```
    UniqueResource(const UniqueResource&) = delete;
    UniqueResource& operator=(const UniqueResource&) = delete;

    void use() const {
        std::cout << "Using unique resource\n";
    }
};

int main() {
    UniqueResource r;
    r.use();

    // UniqueResource copy = r; // error
}
```

16.3.6 Defaulting and deleting beyond constructors

These forms are not limited to constructors. They can apply to special member functions more broadly, including:

- default constructor,
- destructor,
- copy constructor,
- copy assignment operator,
- move constructor,
- move assignment operator.

16.3.7 A practical rule

The special member functions are closely related. If you default or delete one intentionally, you should think carefully about the others as well.

16.4 Member initializer lists

A member initializer list is the part of a constructor that appears after the parameter list and before the function body. It is used to initialize base classes and data members directly.
Example:

```
class Point {
public:
    Point(int x, int y) : x_{x}, y_{y} {}

private:
    int x_;
    int y_;
};
```

Here `x_` and `y_` are initialized directly.

16.4.1 Why member initializer lists matter

They matter because they initialize members directly instead of default-constructing them first and then assigning to them later.
Less preferred style:

```
class Point {
public:
    Point(int x, int y) {
        x_ = x;
        y_ = y;
    }

private:
    int x_;
    int y_;
};
```

Preferred style:

```
class Point {
public:
    Point(int x, int y) : x_{x}, y_{y} {}

private:
    int x_;
    int y_;
};
```

16.4.2 Required cases

Member initializer lists are not just stylistic. They are required in important situations, including:

- reference members,
- const data members,
- members of class type that have no default constructor,
- base-class initialization.

16.4.3 Reference member example

```
class RefHolder {
public:
    explicit RefHolder(int& value) : ref_{value} {}

private:
    int& ref_;
};
```

This must use a member initializer list because a reference must be initialized when created.

16.4.4 Const member example

```
class Config {
public:
    explicit Config(int version) : version_{version} {}

private:
    const int version_;
};
```

A const data member must be initialized directly.

16.4.5 Member construction order

A very important rule is that members are initialized in the order in which they are declared in the class, not in the order written in the initializer list.
Example:

```
class Example {
public:
    Example() : b_{2}, a_{1} {}

private:
    int a_;
    int b_;
};
```

Even though b_ appears first in the initializer list, a_ is initialized first because it is declared first.
This is a critical correctness rule.

16.4.6 A class-type member example

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

class User {
public:
    User(std::string name, int id)
        : name_{std::move(name)}, id_{id} {}

    void print() const {
        std::cout << name_ << " : " << id_ << '\n';
    }

private:
    std::string name_;
    int id_;
};

int main() {
    User u("Ayman", 1001);
```

```
    u.print();
}
```

This is the normal modern way to initialize members.

16.5 Default constructor

A default constructor is a constructor that can be called with no arguments.
Examples:

- a constructor with no parameters,
- a constructor where all parameters have default arguments,
- a compiler-generated default constructor.

16.5.1 User-defined default constructor

```
class Counter {
public:
    Counter() : value_{0} {}

private:
    int value_;
};
```

16.5.2 Compiler-generated default constructor

If the class is eligible, the compiler may generate a default constructor automatically.

```
class Simple {
private:
    int value_{0};
};

int main() {
    Simple s;
}
```

16.5.3 Explicitly defaulted default constructor

```
class Simple {
public:
    Simple() = default;
};
```

16.5.4 When a default constructor may not be generated

A beginner must understand an important rule: if you declare a constructor with parameters, the compiler does not also invent a no-argument constructor for you.
Example:

```
class Person {
public:
    Person(const char* name) : name_{name} {}

private:
    const char* name_;
};

int main() {
    // Person p; // error: no default constructor
    Person p("Ayman");
}
```

16.5.5 Restoring default construction

If you want both a no-argument constructor and a parameterized constructor, you can provide both explicitly.

```
#include <string>

class Person {
public:
    Person() = default;

    explicit Person(const std::string& name)
        : name_{name} {}

private:
    std::string name_{"Unknown"};
};
```

16.5.6 Default constructor with default member initializers

Modern C++ often combines default constructors with in-class member initializers.

```
class Settings {
public:
    Settings() = default;

private:
    int width_{800};
    int height_{600};
    bool fullscreen_{false};
};
```

This is clean and expressive for many classes.

16.5.7 A practical example

```
#include <iostream>

class Window {
public:
```

```
    Window() = default;

    void print() const {
        std::cout << width_ << " x " << height_ << '\n';
    }

private:
    int width_{1024};
    int height_{768};
};

int main() {
    Window w;
    w.print();
}
```

16.6 Introduction to copy and move

C++ objects can often be copied and moved. These operations are controlled by special member functions.
This section is only an introduction. Copy and move semantics will be studied more deeply later, but the student should already understand the basic picture.

16.6.1 Copy construction

A copy constructor creates a new object as a copy of another object of the same type.
Example:

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

class Book {
public:
    Book(const std::string& title) : title_{title} {}

    void print() const {
        std::cout << title_ << '\n';
    }

private:
    std::string title_;
};

int main() {
    Book a("Modern C++");
    Book b = a;

    a.print();
    b.print();
}
```

Here b is copy-constructed from a.

16.6.2 What copying means

Copying generally means the new object receives its own state corresponding to the source object.
For a type like `std::string`, copying creates another string object with the same content.

16.6.3 Move construction

A move constructor creates a new object by taking over resources from another object, usually a temporary or an expiring object.
A simple demonstration with Standard Library behavior:

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

int main() {
    std::string a = "Hello";
    std::string b = std::move(a);

    std::cout << "b = " << b << '\n';
}
```

This introduces the basic idea of moving: transferring resources instead of making an expensive deep copy when appropriate.

16.6.4 Why move exists

Move operations are important because they can make object transfer more efficient for resource-owning types such as:

- strings,
- vectors,
- smart pointers,
- file handles,
- many user-defined resource wrappers.

16.6.5 Special member function family

The special member functions related to copy and move are:

- copy constructor,
- copy assignment operator,
- move constructor,
- move assignment operator.

Together with the destructor and default constructor, they form the main special member function family.

16.6.6 Deleted and defaulted interactions

An important modern rule is that copy and move operations are interrelated. Declaring, deleting, or defining some of them can affect whether others are generated implicitly.
A beginner does not need every detail yet, but must understand this key lesson:

> The special member functions are connected. Changing one often changes the behavior of the others.

16.6.7 A non-copyable example

```
class UniqueHandle {
public:
    UniqueHandle() = default;

    UniqueHandle(const UniqueHandle&) = delete;
    UniqueHandle& operator=(const UniqueHandle&) = delete;
};
```

This expresses that the type should not be copied.

16.6.8 A move-only style example

A full move-only type will be covered later in detail, but the idea looks like this:

```
class MoveOnly {
public:
    MoveOnly() = default;

    MoveOnly(const MoveOnly&) = delete;
    MoveOnly& operator=(const MoveOnly&) = delete;

    MoveOnly(MoveOnly&&) = default;
    MoveOnly& operator=(MoveOnly&&) = default;
};
```

This allows movement but forbids copying.

16.6.9 A practical beginner rule

At this stage, remember these principles:

- copying duplicates state,
- moving transfers resources when appropriate,
- resource-owning classes require careful thought about copy and move behavior,
- defaulted and deleted functions help express the intended semantics clearly.

Conclusion

Constructors and destructors define the life cycle of class objects in C++. Constructors establish a valid initial state. Destructors perform cleanup automatically when an object dies. Together, they are the foundation of RAII and safe resource management. Modern C++ also provides strong control over special member functions:

- `= default` explicitly requests compiler-generated behavior,
- `= delete` explicitly forbids an operation,
- member initializer lists allow direct and often required initialization of members,
- default constructors support no-argument object creation,
- copy and move operations define how objects are duplicated or transferred.

The deeper lesson of this chapter is that object construction is not just about syntax. It is about correctness, invariants, ownership, and lifetime.
A well-designed class uses constructors to guarantee valid objects, destructors to guarantee cleanup, and special member functions to make copying and moving explicit and safe. These ideas will become even more important in the later chapters on RAII, smart pointers, move semantics, and advanced class design.

Encapsulation and Design

Encapsulation is one of the central ideas of Modern C++ class design. It is the practice of grouping data and operations together while carefully controlling what outside code is allowed to see and modify. Good encapsulation protects object state, preserves invariants, clarifies intent, and makes programs easier to maintain.
A beginner often first learns classes as a way to group variables and functions. That is only the starting point. In professional C++ design, a class is not merely a bag of data with attached methods. A class is a controlled abstraction. It defines:

- what users of the type are allowed to do,
- what internal details remain hidden,
- what states are valid,
- how incorrect use is prevented or reduced.

Encapsulation is tightly connected to data hiding, invariants, interface quality, and error prevention. These ideas are not separate topics. They are different aspects of the same design goal: making a type easy to use correctly and difficult to use incorrectly.
This chapter explains:

- data hiding,
- invariants,
- clean interfaces,
- making errors difficult,
- practical design examples that show these ideas in real code.

The goal is not only to write classes that compile. The goal is to write classes that remain correct and useful under real use.

17.1 Data hiding

Data hiding means keeping internal representation details inaccessible to ordinary users of a class. In C++, this is usually achieved through `private` data members and carefully designed `public` member functions.

17.1.1 Why data hiding matters

If internal data is exposed directly, any code can change it freely. That creates several problems:

- the class cannot enforce valid states,
- implementation details leak into outside code,
- changing the representation later becomes harder,
- bugs become easier to introduce because outside code can bypass intended logic.

A weak design exposes the internal state directly:

```
class Temperature {
public:
    double celsius_{0.0};
};
```

This allows arbitrary writes:

```
int main() {
    Temperature t;
    t.celsius_ = -1000.0;
}
```

Whether such a value is meaningful depends on the intended domain. The main issue is that the class has no control.

17.1.2 Hiding data behind an interface

A better design hides the data and exposes controlled operations:

```
class Temperature {
public:
    void set_celsius(double value) {
        celsius_ = value;
    }

    double celsius() const {
        return celsius_;
    }

private:
    double celsius_{0.0};
};
```

Now the internal representation is hidden. The class can later add validation, logging, conversion logic, or unit handling without forcing all calling code to change.

17.1.3 Data hiding protects representation choices

Suppose a class initially stores a person's age as an `int`:

```
class Person {
public:
    void set_age(int age) {
        age_ = age;
    }

    int age() const {
        return age_;
    }

private:
    int age_{0};
};
```

If the data member is private, the implementation can later change:

- age may be stored differently,
- validation may be added,
- related metadata may be tracked,
- units may be changed,
- calculations may be cached.

If the data were public, all external code would depend directly on the internal representation.

17.1.4 A practical example

Weak design:

```
class BankAccount {
public:
    double balance_{0.0};
};
```

Better design:

```
class BankAccount {
public:
    void deposit(double amount) {
        if (amount > 0.0) {
            balance_ += amount;
        }
    }

    bool withdraw(double amount) {
        if (amount <= 0.0 || amount > balance_) {
            return false;
```

```
        }

        balance_ -= amount;
        return true;
    }

    double balance() const {
        return balance_;
    }

private:
    double balance_{0.0};
};
```

This design hides the balance representation and forces interaction through meaningful operations.

17.1.5 Data hiding is not secrecy for its own sake

Data hiding is not about making code mysterious. It is about making responsibilities clear.
A public interface should answer:

- what this object can do,
- what information can be observed,
- what modifications are valid.

The implementation answers:

- how the object stores its data,
- how it preserves correctness,
- how it performs its work internally.

17.2 Invariants

An invariant is a condition that should always hold for a valid object, except possibly during carefully controlled internal transitions. Preserving invariants is one of the most important responsibilities of class design.

17.2.1 What an invariant means

If a class represents a meaningful concept, not every possible combination of member values should necessarily be valid.
Examples of possible invariants:

- A bank account balance must not be negative.
- A date must have a valid month and day.
- A rectangle's width and height must not be negative.
- An index range must satisfy `begin <= end`.
- A non-empty identifier must contain at least one character.

17.2.2 A class with weak invariant control

```
class Rectangle {
public:
    int width_{0};
    int height_{0};
};
```

Now outside code may create invalid states freely:

```
int main() {
    Rectangle r;
    r.width_ = -10;
    r.height_ = -20;
}
```

17.2.3 A class that enforces invariants

```
class Rectangle {
public:
    void set_width(int width) {
        if (width >= 0) {
            width_ = width;
        }
    }

    void set_height(int height) {
        if (height >= 0) {
            height_ = height;
        }
    }

    int width() const {
        return width_;
    }

    int height() const {
        return height_;
    }

    int area() const {
        return width_ * height_;
    }

private:
    int width_{0};
    int height_{0};
};
```

This class protects the invariant that width and height remain non-negative.

17.2.4 Constructors and invariants

Constructors are a natural place to establish invariants at object creation time.

```
class Percentage {
public:
    explicit Percentage(int value)
        : value_{(value < 0) ? 0 : (value > 100 ? 100 : value)} {}

    int value() const {
        return value_;
    }

private:
    int value_;
};
```

This ensures that the stored value stays in the range from 0 to 100.

17.2.5 Invariant-preserving operations

Every public function should preserve the class invariants. This is one of the strongest practical tests of good class design.
Consider a counter that must never go below zero:

```
class Counter {
public:
    void increment() {
        ++value_;
    }

    void decrement() {
        if (value_ > 0) {
            --value_;
        }
    }

    int value() const {
        return value_;
    }

private:
    int value_{0};
};
```

The public operations preserve the invariant that the count remains non-negative.

17.2.6 Why invariants matter

Invariants matter because they reduce the number of states programmers must reason about. If a class guarantees its own correctness conditions, users of the class can rely on those guarantees.
Without invariants:

- every caller must defend against invalid internal state,
- assumptions become fragile,
- debugging becomes harder.

With invariants:

- the class becomes more trustworthy,
- the rest of the code becomes simpler,
- reasoning about the program becomes easier.

17.3 Clean interfaces

A clean interface is an interface that is easy to understand, hard to misuse, and focused on meaningful operations rather than implementation details.

17.3.1 What makes an interface clean

A clean interface usually has these qualities:

- the names are clear,
- each operation has a specific purpose,
- the public surface is minimal but sufficient,
- invalid usage is reduced,
- the representation stays hidden,
- the interface reflects the domain concept rather than internal mechanics.

17.3.2 A cluttered interface

```
class UserRecord {
public:
    std::string first_name_;
    std::string last_name_;
    int age_;
    bool active_;
};
```

This class exposes everything directly. It gives no guidance about what operations make sense, what combinations are valid, or how state should be managed.

17.3.3 A cleaner interface

```cpp
#include <string>

class UserRecord {
public:
    void set_name(const std::string& first, const std::string& last) {
        first_name_ = first;
        last_name_ = last;
    }

    void set_age(int age) {
        if (age >= 0) {
            age_ = age;
        }
    }

    void activate() {
        active_ = true;
    }

    void deactivate() {
        active_ = false;
    }

    std::string full_name() const {
        return first_name_ + " " + last_name_;
    }

    int age() const {
        return age_;
    }

    bool active() const {
        return active_;
    }

private:
    std::string first_name_;
    std::string last_name_;
    int age_{0};
    bool active_{false};
};
```

This version exposes useful behavior rather than raw internal storage.

17.3.4 Prefer operations over exposed state

A strong interface asks:

> What should users of this type be allowed to do?

Instead of:

Which fields should I expose?

For example, a stack-like abstraction should expose operations such as:

- push,
- pop,
- top,
- empty.

It should not necessarily expose the internal container directly.

17.3.5 Const-correct interfaces

A clean interface also uses `const` correctly for read-only operations.

```
class Counter {
public:
    void increment() {
        ++value_;
    }

    int value() const {
        return value_;
    }

private:
    int value_{0};
};
```

Marking read-only operations as `const` helps both correctness and readability.

17.3.6 Keep the public surface small

Every public member becomes part of the class contract. Public members are promises to users of the class.
That means a well-designed class should avoid exposing more than necessary. A smaller interface:

- is easier to learn,
- is easier to document,
- is easier to preserve over time,
- gives fewer opportunities for misuse.

17.3.7 Prefer meaningful names

Good names improve interface quality dramatically.
Weak names:

```
class FileThing {
public:
    void do_it();
    int x() const;
};
```

Better names:

```
class FileReader {
public:
    bool open(const std::string& path);
    bool read_line(std::string& out);
    bool is_open() const;
};
```

The second interface communicates purpose much more clearly.

17.4 Making errors difficult

One of the strongest goals of encapsulation and good design is to make incorrect use difficult. In Modern C++, a good type should guide users toward correct code.

17.4.1 The principle

A useful design principle is:

> Make the correct way easy and the incorrect way hard.

This does not mean every error becomes impossible, but a well-designed interface can reduce many common mistakes.

17.4.2 Hide dangerous operations

If a class has operations that can break invariants or violate ownership rules, they should not be exposed casually.
For example, if a bank account should not allow arbitrary balance assignment, the class should not expose `set_balance()` unless that operation is semantically valid.
Weak design:

```
class BankAccount {
public:
    void set_balance(double value) {
        balance_ = value;
    }

    double balance() const {
        return balance_;
    }

private:
    double balance_{0.0};
};
```

This allows invalid changes too easily.

Better design:

```
class BankAccount {
public:
    void deposit(double amount) {
        if (amount > 0.0) {
            balance_ += amount;
        }
    }

    bool withdraw(double amount) {
        if (amount <= 0.0 || amount > balance_) {
            return false;
        }

        balance_ -= amount;
        return true;
    }

    double balance() const {
        return balance_;
    }

private:
    double balance_{0.0};
};
```

The second design makes invalid balance manipulation much harder.

17.4.3 Use constructors to prevent incomplete objects

Constructors can prevent partially initialized or inconsistent objects.

Weak style:

```
class ConnectionInfo {
public:
    std::string host_;
    int port_;
};
```

This allows creation of objects without meaningful state.

Stronger style:

```
#include <string>

class ConnectionInfo {
public:
    ConnectionInfo(std::string host, int port)
        : host_{std::move(host)}, port_{port > 0 ? port : 1} {}

    const std::string& host() const {
```

```
        return host_;
    }

    int port() const {
        return port_;
    }

private:
    std::string host_;
    int port_;
};
```

Now each object starts life with a proper host and a controlled port value.

17.4.4 Use return values to communicate failure

A class interface should make failure visible when failure is part of the domain.

```
class Counter {
public:
    bool decrement() {
        if (value_ == 0) {
            return false;
        }

        --value_;
        return true;
    }

    void increment() {
        ++value_;
    }

    int value() const {
        return value_;
    }

private:
    int value_{0};
};
```

Returning `bool` here makes failure explicit.

17.4.5 Use `const` to prevent accidental mutation

Read-only operations should be marked `const`. This helps prevent accidental state modification and allows the type to be used in more contexts.

```
class Name {
public:
    const std::string& get() const {
        return value_;
```

```
    }

private:
    std::string value_;
};
```

17.4.6 Delete invalid operations when necessary

Sometimes the best way to make errors difficult is to forbid an operation completely.

```
class UniqueResource {
public:
    UniqueResource() = default;

    UniqueResource(const UniqueResource&) = delete;
    UniqueResource& operator=(const UniqueResource&) = delete;
};
```

This prevents accidental copying of a type whose semantics should remain unique.

17.4.7 Prefer meaningful abstractions over raw access

If callers must constantly remember low-level rules, the design is probably too weak.
Compare:

- a class that exposes raw internal pointers and expects the caller to manage everything,
- a class that exposes clear domain-level actions and manages internals safely.

The second design is usually more robust.

17.5 Practical design examples

This section puts the principles together through practical class examples.

17.5.1 Example 1: a validated percentage type

A percentage value should stay in the range from 0 to 100.
Weak design:

```
class Percentage {
public:
    int value_;
};
```

Better design:

```
#include <iostream>

class Percentage {
public:
```

```
    explicit Percentage(int value)
        : value_{clamp(value)} {}

    void set(int value) {
        value_ = clamp(value);
    }

    int get() const {
        return value_;
    }

    void print() const {
        std::cout << value_ << "%\n";
    }

private:
    static int clamp(int value) {
        if (value < 0) {
            return 0;
        }
        if (value > 100) {
            return 100;
        }
        return value;
    }

    int value_;
};

int main() {
    Percentage p(150);
    p.print();

    p.set(-20);
    p.print();
}
```

This design:

- hides the representation,
- preserves the invariant,
- makes invalid percentage values difficult.

17.5.2 Example 2: a task status type

A task should have a title and a controlled completion state.

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

class Task {
```

```
public:
    explicit Task(std::string title)
        : title_{std::move(title)} {}

    void mark_done() {
        done_ = true;
    }

    void mark_pending() {
        done_ = false;
    }

    const std::string& title() const {
        return title_;
    }

    bool done() const {
        return done_;
    }

    void print() const {
        std::cout << "[" << (done_ ? "done" : "pending")
                  << "] " << title_ << '\n';
    }

private:
    std::string title_;
    bool done_{false};
};

int main() {
    Task t("Write chapter on encapsulation");
    t.print();

    t.mark_done();
    t.print();
}
```

This design:

- keeps state private,
- exposes meaningful operations,
- prevents arbitrary external manipulation of internal flags.

17.5.3 Example 3: a range with an invariant

A range should satisfy `begin <= end`.

```
#include <iostream>

class IntRange {
```

```
public:
    IntRange(int begin, int end)
        : begin_{begin}, end_{end} {
        normalize();
    }

    void set_begin(int begin) {
        begin_ = begin;
        normalize();
    }

    void set_end(int end) {
        end_ = end;
        normalize();
    }

    int begin() const {
        return begin_;
    }

    int end() const {
        return end_;
    }

    int length() const {
        return end_ - begin_;
    }

private:
    void normalize() {
        if (begin_ > end_) {
            int temp = begin_;
            begin_ = end_;
            end_ = temp;
        }
    }

    int begin_;
    int end_;
};

int main() {
    IntRange r(10, 3);

    std::cout << r.begin() << " " << r.end()
              << " length=" << r.length() << '\n';
}
```

This class preserves its own invariant automatically.

17.5.4 Example 4: a cleaner counter interface

Weak design:

```
class Counter {
public:
    int value_{0};
};
```

Better design:

```
#include <iostream>

class Counter {
public:
    void increment() {
        ++value_;
    }

    bool decrement() {
        if (value_ == 0) {
            return false;
        }

        --value_;
        return true;
    }

    void reset() {
        value_ = 0;
    }

    int value() const {
        return value_;
    }

private:
    int value_{0};
};

int main() {
    Counter c;

    c.increment();
    c.increment();
    c.decrement();

    std::cout << c.value() << '\n';
}
```

This version:

- hides the data,

- preserves non-negative state,
- presents operations in domain terms.

17.5.5 Example 5: interface separated from implementation

Header-like interface:

```
// account.hpp
#pragma once

class Account {
public:
    explicit Account(double initial_balance);

    void deposit(double amount);
    bool withdraw(double amount);
    double balance() const;

private:
    double balance_;
};
```

Implementation-like source:

```
// account.cpp
#include "account.hpp"

Account::Account(double initial_balance)
    : balance_{initial_balance < 0.0 ? 0.0 : initial_balance} {}

void Account::deposit(double amount) {
    if (amount > 0.0) {
        balance_ += amount;
    }
}

bool Account::withdraw(double amount) {
    if (amount <= 0.0 || amount > balance_) {
        return false;
    }

    balance_ -= amount;
    return true;
}

double Account::balance() const {
    return balance_;
}
```

This illustrates a classic encapsulation pattern:

- the interface tells users what the class does,

- the implementation contains the internal logic,
- the data remains hidden.

Conclusion

Encapsulation is not an optional stylistic extra in Modern C++. It is one of the main ways to build trustworthy types.
The core ideas of this chapter work together:

- data hiding protects internal representation,
- invariants keep objects valid,
- clean interfaces expose only meaningful operations,
- good design makes errors difficult,
- practical classes become easier to use correctly and harder to misuse.

A well-encapsulated class does more than store data. It defines a controlled abstraction with clear responsibilities and reliable behavior.
In practical Modern C++ design, a strong default approach is:

- keep data members private,
- expose a minimal and meaningful public interface,
- establish and preserve invariants from construction onward,
- design operations so that invalid states and invalid actions are reduced by construction.

This approach leads to code that is easier to maintain, easier to reason about, and more resistant to accidental misuse. It is one of the main steps from simply writing classes toward designing good C++ types.

Inheritance and Polymorphism

Inheritance and polymorphism are central concepts in object-oriented programming, but in Modern C++ they must be used carefully and with clear intent. While earlier C++ styles often relied heavily on inheritance, modern design emphasizes correctness, clarity, and maintainability over excessive hierarchy.
Inheritance allows one class to reuse and extend another. Polymorphism allows different types to be treated uniformly through a common interface. Together, they enable flexible and extensible designs, especially when working with abstract interfaces and runtime behavior.
This chapter explains:

- how inheritance works,
- how virtual functions enable polymorphism,
- the role of `override` and `final`,
- abstract classes,
- when inheritance is appropriate,
- composition as an alternative,
- common mistakes and how to avoid them.

The goal is not only to understand syntax, but to understand when and why to use these features correctly in Modern C++.

18.1 Inheritance

Inheritance allows a class (derived class) to reuse and extend the behavior of another class (base class).
Basic syntax:

```
class Base {
public:
    void greet() const {
        std::cout << "Hello from Base\n";
    }
};

class Derived : public Base {
};
```

18.1.1 Using a derived class

```
#include <iostream>

int main() {
    Derived d;
    d.greet();
}
```

The derived class inherits public members of the base class.

18.1.2 Access control in inheritance

The inheritance specifier controls how base class members are accessed:

- `public` inheritance: public remains public, protected remains protected
- `protected` inheritance: public becomes protected
- `private` inheritance: public becomes private

Most polymorphic designs use `public` inheritance.

18.1.3 Extending a base class

```
#include <iostream>

class Animal {
public:
    void eat() const {
        std::cout << "Animal eats\n";
    }
};

class Dog : public Animal {
public:
    void bark() const {
        std::cout << "Dog barks\n";
    }
};

int main() {
    Dog d;
    d.eat();
    d.bark();
}
```

18.1.4 The “is-a” relationship

Inheritance models an **is-a** relationship:

- a Dog is an Animal,

- a Circle is a Shape,
- a FileLogger is a Logger.

If this relationship is not true, inheritance is likely the wrong tool.

18.2 Virtual functions

Virtual functions enable runtime polymorphism. They allow a derived class to provide a specific implementation of a function declared in a base class.

18.2.1 Basic virtual function

```
#include <iostream>

class Shape {
public:
    virtual void draw() const {
        std::cout << "Drawing Shape\n";
    }
};

class Circle : public Shape {
public:
    void draw() const {
        std::cout << "Drawing Circle\n";
    }
};
```

18.2.2 Polymorphic behavior

```
int main() {
    Shape* s = new Circle();
    s->draw();
    delete s;
}
```

Even though the pointer type is `Shape*`, the `Circle` implementation is called. This is runtime polymorphism.

18.2.3 Why virtual functions matter

Virtual functions allow:

- writing code against interfaces,
- substituting different implementations,
- extending systems without modifying existing code.

18.2.4 Virtual destructor

A base class intended for polymorphic use must have a virtual destructor.

```
class Shape {
public:
    virtual ~Shape() = default;
};
```

Without a virtual destructor, deleting derived objects through base pointers leads to undefined behavior.

18.3 `override` and `final`

Modern C++ introduces `override` and `final` to improve correctness and clarity.

18.3.1 `override`

`override` ensures that a function actually overrides a virtual function from the base class.

```
class Shape {
public:
    virtual void draw() const {}
};

class Circle : public Shape {
public:
    void draw() const override {
    }
};
```

If the signature does not match, the compiler produces an error.

18.3.2 Why `override` is important

Without `override`, subtle mistakes may occur:

```
class Shape {
public:
    virtual void draw() const {}
};

class Circle : public Shape {
public:
    void draw() { } // different signature
};
```

This does not override the base function. Using `override` prevents such errors.

18.3.3 `final`

`final` prevents further overriding or inheritance.

```
class Shape {
public:
    virtual void draw() const {}
};

class Circle final : public Shape {
public:
    void draw() const override {}
};
```

Now `Circle` cannot be inherited.

18.3.4 Final virtual function

```
class Base {
public:
    virtual void process() const final {}
};
```

Derived classes cannot override `process()`.

18.4 Abstract classes

An abstract class is a class that cannot be instantiated and is used only as a base class.

18.4.1 Pure virtual function

A pure virtual function is declared using `= 0`.

```
class Shape {
public:
    virtual void draw() const = 0;
    virtual ~Shape() = default;
};
```

18.4.2 Derived implementation

```
#include <iostream>

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing Circle\n";
    }
};
```

18.4.3 Usage

```
int main() {
    Shape* s = new Circle();
    s->draw();
    delete s;
}
```

18.4.4 Why abstract classes matter

Abstract classes define interfaces:

- they separate interface from implementation,
- they allow multiple implementations,
- they support extensibility.

18.5 When to use inheritance

Inheritance should be used when:

- there is a clear **is-a** relationship,
- a common interface is needed,
- polymorphic behavior is required,
- code reuse aligns with logical hierarchy.

Example:

```
class Logger {
public:
    virtual void log(const std::string& msg) = 0;
    virtual ~Logger() = default;
};

class FileLogger : public Logger {
public:
    void log(const std::string& msg) override {
    }
};

class ConsoleLogger : public Logger {
public:
    void log(const std::string& msg) override {
    }
};
```

18.6 Composition vs inheritance

Modern C++ strongly prefers composition over inheritance in many cases.

18.6.1 Composition

Composition means building classes using other classes as members.

```
#include <iostream>

class Engine {
public:
    void start() const {
        std::cout << "Engine started\n";
    }
};

class Car {
public:
    void start() const {
        engine_.start();
    }

private:
    Engine engine_;
};
```

18.6.2 Why composition is often better

Composition:

- avoids tight coupling,
- is easier to modify,
- avoids fragile base class problems,
- models “has-a” relationships naturally.

18.6.3 Inheritance vs composition

- Inheritance: “is-a”
- Composition: “has-a”

Incorrect inheritance example:

```
class Engine {
};

class Car : public Engine { // incorrect
};
```

Correct composition:

```
class Car {
    Engine engine_;
};
```

18.6.4 A flexible composition example

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

class Renderer {
public:
    virtual void render() const = 0;
    virtual ~Renderer() = default;
};

class OpenGLRenderer : public Renderer {
public:
    void render() const override {
        std::cout << "OpenGL rendering\n";
    }
};

class VulkanRenderer : public Renderer {
public:
    void render() const override {
        std::cout << "Vulkan rendering\n";
    }
};

class Game {
public:
    explicit Game(std::unique_ptr<Renderer> r)
        : renderer_{std::move(r)} {}

    void draw() const {
        renderer_->render();
    }

private:
    std::unique_ptr<Renderer> renderer_;
};
```

18.7 Common mistakes

18.7.1 Missing virtual destructor

```
class Base {
public:
    ~Base() {} // wrong if used polymorphically
};
```

Fix:

```
class Base {
public:
    virtual ~Base() = default;
```

```
};
```

18.7.2 Forgetting `override`

Failing to use `override` may introduce subtle bugs.

18.7.3 Using inheritance for code reuse only

Inheritance should not be used just to reuse code. It should represent a real conceptual relationship.

18.7.4 Breaking invariants in derived classes

Derived classes must respect base class expectations.

18.7.5 Exposing too much in base classes

A base class should expose only what is necessary. Excessive public members make the hierarchy fragile.

18.7.6 Deep inheritance hierarchies

Deep hierarchies are hard to understand and maintain. Prefer shallow hierarchies and composition.

18.7.7 Object slicing

```
class Base {
public:
    virtual void f() const {}
};

class Derived : public Base {
public:
    void f() const override {}
};

int main() {
    Derived d;
    Base b = d; // slicing
}
```

The derived part is lost. Use references or pointers instead.

18.7.8 Incorrect polymorphic usage

```
Base b = Derived{};
b.f();
```

This does not use polymorphism. Use:

```
Base& b = d;
b.f();
```

Conclusion

Inheritance and polymorphism are powerful but must be used carefully in Modern C++.
Key ideas:

- inheritance models “is-a” relationships,
- virtual functions enable runtime polymorphism,
- `override` and `final` improve safety,
- abstract classes define interfaces,
- composition is often a better alternative,
- misuse leads to fragile and complex designs.

A modern C++ developer uses inheritance deliberately, not automatically. The focus is on correctness, clarity, and maintainability rather than building deep hierarchies.
Understanding these principles is essential for designing robust and extensible systems.

Part VI

The Core of C++: RAII and Resource Management

Resource Management

19.1 What is a resource

In C++, a *resource* is anything that must be acquired before use and released correctly after use. Memory is the most famous example, but it is only one example. In real software, especially on Windows, many important objects are resources even though they are not ordinary C++ objects. A file opened from disk, a synchronization primitive, a socket, a system handle, a database connection, a mapped view of memory, a GDI object, or a COM interface pointer all represent something that the program must manage carefully.
A useful way to think about a resource is this:

- It has an **ownership question**: who is responsible for it?
- It has a **lifetime question**: when does it become valid, and when must it stop being used?
- It has a **cleanup requirement**: what exact action must happen when the resource is no longer needed?

A plain integer usually is not a resource. It does not require explicit cleanup. A raw pointer by itself is not necessarily a resource either; it may merely observe another object. But dynamically allocated memory obtained with `new`, a file stream connected to an actual file, or a Windows `HANDLE` are all resources because they require explicit release.
The central problem in systems programming is not acquiring resources. Acquisition is usually easy. The difficult part is guaranteeing release in every execution path:

- normal return
- early return
- exception
- partial construction
- failure in the middle of a function

C++ solves this problem through deterministic object lifetime. When an automatic object leaves scope, its destructor runs immediately. This is the foundation on which modern C++ resource management is built.

19.1.1 Resources are broader than memory

Beginners often associate resource management only with heap memory. That view is too narrow. Modern C++ treats many kinds of external or limited entities as resources:

- dynamically allocated memory
- files and file descriptors
- Windows kernel handles
- mutexes and locks
- sockets
- threads that must be joined or detached
- database transactions
- temporary directories or files
- graphics objects
- mapped memory regions

Each of these has the same design challenge: acquisition and release must stay paired, even when the code becomes large or exceptions occur.

19.1.2 Ownership is the key idea

When talking about resources, the most important question is ownership. If ownership is unclear, bugs become likely. The program may leak a resource, release it too early, or release it twice.
A strong Modern C++ design tries to make one of these situations explicit:

- **exclusive ownership**: exactly one object owns the resource
- **shared ownership**: multiple objects cooperate in lifetime management
- **non-owning access**: an object may use the resource, but does not destroy it

The standard library reflects these cases through types such as `std::unique_ptr`, `std::shared_ptr`, and raw/reference-style observers.

19.2 Files, memory, handles, locks

This section introduces common resource categories and shows why each one needs disciplined management.

19.2.1 Memory

Dynamic memory is acquired explicitly and must eventually be released. In old C-style or early C++ code, this often appeared as a manual pair of operations:

```
int* p = new int(42);
// use p
delete p;
```

This style is fragile because every exit path must remember to call `delete`. If an exception is thrown after allocation but before deletion, the memory leaks.

Problematic manual memory management

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

void process(bool fail)
{
    int* data = new int[100];

    if (fail)
        throw std::runtime_error("processing failed");

    std::cout << data[0] << '\n';
    delete[] data;
}
```

If `fail` is true, the array is never released. The code works only in the success path.

RAII-based memory management

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

void process(bool fail)
{
    auto data = std::make_unique<int[]>(100);

    if (fail)
        throw std::runtime_error("processing failed");

    std::cout << data[0] << '\n';
}
```

Now the array is released automatically when `data` leaves scope. No explicit `delete[]` is needed.

19.2.2 Files

A file is a classic resource. Opening succeeds or fails, and once opened, the file must be closed. Fortunately, the standard library already models file streams as RAII types.

Reading from a file safely

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

void read_config()
{
    std::ifstream in("config.txt");
    if (!in)
    {
```

```
        std::cerr << "Failed to open config.txt\n";
        return;
    }

    std::string line;
    while (std::getline(in, line))
    {
        std::cout << line << '\n';
    }
}
```

The important idea is not only that `std::ifstream` can read a file. The important design point is that the file closes automatically when the stream object is destroyed. That remains true for normal return, early return, and exception unwinding.

Writing to a file safely

```
#include <fstream>
#include <stdexcept>
#include <string>

void save_log(const std::string& text)
{
    std::ofstream out("app.log", std::ios::app);
    if (!out)
        throw std::runtime_error("cannot open app.log");

    out << text << '\n';
}
```

Again, the release operation is bound to object lifetime.

19.2.3 Windows handles

On Windows, many operating system resources are represented by `HANDLE`. A handle is not self-managing. If acquired from an API such as `CreateFileW`, it must be released with the correct function, often `CloseHandle`. This is exactly the kind of resource that benefits from a small RAII wrapper.

Danger of manual handle management

```
#include <windows.h>
#include <stdexcept>

void manual_handle_example()
{
    HANDLE hFile = CreateFileW(
        L"example.txt",
        GENERIC_READ,
        FILE_SHARE_READ,
        nullptr,
        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL,
```

```
        nullptr
    );

    if (hFile == INVALID_HANDLE_VALUE)
        throw std::runtime_error("CreateFileW failed");

    // Suppose more code here can throw.

    CloseHandle(hFile);
}
```

If code between acquisition and `CloseHandle` throws, the handle leaks.

A simple RAII wrapper for HANDLE

```
#include <windows.h>
#include <stdexcept>
#include <utility>

class unique_handle
{
private:
    HANDLE handle_{INVALID_HANDLE_VALUE};

public:
    unique_handle() noexcept = default;

    explicit unique_handle(HANDLE h) noexcept
        : handle_(h)
    {
    }

    ~unique_handle()
    {
        if (handle_ != nullptr && handle_ != INVALID_HANDLE_VALUE)
        {
            CloseHandle(handle_);
        }
    }

    unique_handle(const unique_handle&) = delete;
    unique_handle& operator=(const unique_handle&) = delete;

    unique_handle(unique_handle&& other) noexcept
        : handle_(std::exchange(other.handle_, INVALID_HANDLE_VALUE))
    {
    }

    unique_handle& operator=(unique_handle&& other) noexcept
    {
        if (this != &other)
        {
            reset();
```

```
            handle_ = std::exchange(other.handle_, INVALID_HANDLE_VALUE);
        }
        return *this;
    }

    HANDLE get() const noexcept
    {
        return handle_;
    }

    explicit operator bool() const noexcept
    {
        return handle_ != nullptr && handle_ != INVALID_HANDLE_VALUE;
    }

    void reset(HANDLE new_handle = INVALID_HANDLE_VALUE) noexcept
    {
        if (handle_ != nullptr && handle_ != INVALID_HANDLE_VALUE)
        {
            CloseHandle(handle_);
        }
        handle_ = new_handle;
    }

    HANDLE release() noexcept
    {
        return std::exchange(handle_, INVALID_HANDLE_VALUE);
    }
};

void safe_handle_example()
{
    unique_handle hFile{
        CreateFileW(
            L"example.txt",
            GENERIC_READ,
            FILE_SHARE_READ,
            nullptr,
            OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL,
            nullptr
        )
    };

    if (!hFile)
        throw std::runtime_error("CreateFileW failed");

    // Any exception after this point still leads to automatic cleanup.
}
```

This wrapper expresses exclusive ownership. It also shows an important rule of modern resource types: copy is deleted, move is allowed.

19.2.4 Locks

A lock is a resource representing temporary ownership of synchronized access. The lock must be released exactly once and at the correct time. Manual lock and unlock code is dangerous because an exception can bypass the unlock call.

Incorrect manual locking style

```
#include <iostream>
#include <mutex>
#include <stdexcept>

std::mutex g_mutex;

void risky_operation(bool fail)
{
    g_mutex.lock();

    if (fail)
        throw std::runtime_error("operation failed");

    std::cout << "protected work\n";
    g_mutex.unlock();
}
```

If an exception is thrown, the mutex remains locked. That can block other threads indefinitely.

Correct RAII locking with std::lock_guard

```
#include <iostream>
#include <mutex>
#include <stdexcept>

std::mutex g_mutex;

void safe_operation(bool fail)
{
    std::lock_guard<std::mutex> lock(g_mutex);

    if (fail)
        throw std::runtime_error("operation failed");

    std::cout << "protected work\n";
}
```

The destructor of `std::lock_guard` unlocks the mutex automatically at scope exit.

Flexible RAII locking with std::unique_lock

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>
```

```
std::mutex g_mutex;

void unique_lock_example()
{
    std::unique_lock<std::mutex> lock(g_mutex, std::defer_lock);

    // Do some non-protected work here first.

    lock.lock();
    std::cout << "critical section\n";
    lock.unlock();

    // More non-protected work can continue here.
}
```

`std::unique_lock` is heavier than `std::lock_guard`, but it offers deferred locking, unlocking before scope ends, and transfer of lock ownership.

19.2.5 Other resources with the same pattern

Once the idea is understood, the same design applies everywhere:

- a thread resource can be joined through an owning wrapper
- a database transaction can commit or roll back through a guard object
- a temporary file can be deleted in a destructor
- a mapped view can unmap itself automatically

This is why RAII is not a small technique. It is a general lifetime discipline.

19.3 RAII concept

RAII stands for **Resource Acquisition Is Initialization**. The name can sound unusual at first, but the idea is precise and powerful.
A resource is acquired during object construction and released during object destruction. The object therefore becomes the owner of the resource. As long as the object is alive, the resource is valid. When the object dies, the resource is released automatically.
This model connects resource lifetime to object lifetime.

19.3.1 Core definition

A type follows RAII when it has these properties:

- The constructor establishes ownership of a valid resource or reports failure.
- The destructor releases the owned resource.

- The object enforces an invariant such as “either I own a valid resource, or I am empty”.
- Cleanup does not depend on callers remembering to perform manual release.

That last point is the main victory. Instead of asking humans to pair acquire and release correctly in every code path, RAII lets the language runtime enforce the cleanup through scope exit.

19.3.2 A minimal custom RAII type

```
#include <cstdio>
#include <stdexcept>

class file_wrapper
{
private:
    std::FILE* file_{nullptr};

public:
    explicit file_wrapper(const char* path, const char* mode)
        : file_(std::fopen(path, mode))
    {
        if (!file_)
            throw std::runtime_error("std::fopen failed");
    }

    ~file_wrapper()
    {
        if (file_)
            std::fclose(file_);
    }

    file_wrapper(const file_wrapper&) = delete;
    file_wrapper& operator=(const file_wrapper&) = delete;

    file_wrapper(file_wrapper&& other) noexcept
        : file_(other.file_)
    {
        other.file_ = nullptr;
    }

    file_wrapper& operator=(file_wrapper&& other) noexcept
    {
        if (this != &other)
        {
            if (file_)
                std::fclose(file_);
            file_ = other.file_;
            other.file_ = nullptr;
        }
        return *this;
    }
```

```
    std::FILE* get() const noexcept
    {
        return file_;
    }
};
```

This class demonstrates the essential structure of RAII:

- acquire in constructor
- release in destructor
- disallow accidental copying
- allow transfer through moves

19.3.3 RAII and invariants

A well-designed RAII class maintains a strong invariant. For example, the wrapper above tries to guarantee this rule:

> If a `file_wrapper` object exists successfully, then it owns a valid `FILE*`. After a move, the source becomes empty and harmless.

This invariant simplifies the rest of the program. Functions using the object do not need separate cleanup logic and do not need to guess whether ownership is shared.

19.3.4 RAII is not garbage collection

RAII is sometimes misunderstood as if it were the same as automatic memory management in garbage-collected languages. It is not.
RAII is deterministic. Destruction occurs exactly at scope exit for automatic objects, or when the owning object itself is destroyed. This deterministic timing is extremely valuable in systems programming because locks, files, handles, and transactions often must be released at a precise moment, not at some later time chosen by a runtime collector.

19.3.5 RAII and exceptions

RAII becomes even more important when exceptions are used. During stack unwinding, destructors of fully constructed automatic objects are called in reverse order of construction. This means each RAII object still performs cleanup even when the function exits abnormally.

```
#include <fstream>
#include <memory>
#include <mutex>
#include <stdexcept>
#include <string>

std::mutex g_mutex;

void combined_example()
{
```

```
    std::lock_guard<std::mutex> lock(g_mutex);
    std::ifstream in("data.txt");
    auto buffer = std::make_unique<char[]>(4096);

    if (!in)
        throw std::runtime_error("cannot open data.txt");

    throw std::runtime_error("unexpected failure");
}
```

Even though the function throws, cleanup still happens correctly:

- `buffer` releases memory
- `in` closes the file
- `lock` unlocks the mutex

That is the practical beauty of RAII.

19.4 Why RAII is fundamental

RAII is not merely one feature among many. It is one of the deepest organizing principles in C++. Many modern library types, idioms, and best practices either directly implement RAII or depend on it.

19.4.1 It makes cleanup automatic and local

Without RAII, resource cleanup is manual and scattered. With RAII, cleanup is automatic and local to the object that owns the resource. This keeps acquisition and release logic together.
Compare the mental models:
Manual style:

- acquire here
- remember all exits
- release later
- hope no path was forgotten

RAII style:

- create owner object
- use it normally
- let the destructor clean up

The second model scales much better.

19.4.2 It gives exception safety

A major reason modern C++ prefers RAII is exception safety. If code uses raw resources directly, exceptions can interrupt control flow and skip manual release code. RAII converts cleanup into ordinary destruction, and destruction is part of the language lifetime model.
This is why modern guidance strongly prefers:

- `std::unique_ptr` over raw owning pointers
- `std::lock_guard` or `std::unique_lock` over manual `lock`/`unlock`
- stream classes and wrapper types over manual cleanup logic

19.4.3 It reduces leaks and double releases

Resource bugs often take one of these forms:

- forgetting to release
- releasing too early
- releasing twice
- using a resource after release

RAII does not magically eliminate every possible bug, but it removes a large class of such errors by making ownership explicit and by centralizing release in one destructor.

19.4.4 It improves class design

RAII encourages better interfaces. Instead of passing raw low-level resources around and hoping every function behaves correctly, code can pass owning or non-owning abstractions with clear semantics.
For example, compare these two styles.
Weak design:

```
void process_file(std::FILE* f);
```

Who owns `f`? Who closes it? The interface does not say.
Stronger design:

```
void process_file(file_wrapper& f);
```

The ownership policy is much clearer: the caller has an already-managed object, and the function uses it without taking over destruction.

19.4.5 It fits the standard library naturally

Modern C++ is full of RAII-based types:

- `std::string`
- `std::vector`
- `std::unique_ptr`
- `std::shared_ptr`
- `std::lock_guard`
- `std::unique_lock`
- `std::ifstream`, `std::ofstream`, `std::fstream`
- many platform wrappers and library adapters

A C++ programmer who understands RAII understands why these classes are shaped the way they are.

19.4.6 It enables composability

One of the strongest practical benefits of RAII is composability. A function can safely own several resources at once by storing each resource in its own RAII object.

```
#include <fstream>
#include <memory>
#include <mutex>
#include <string>

class logger
{
private:
    std::mutex mutex_;
    std::ofstream file_;

public:
    explicit logger(const std::string& path)
        : file_(path, std::ios::app)
    {
        if (!file_)
            throw std::runtime_error("failed to open log file");
    }

    void write(const std::string& message)
    {
        std::lock_guard<std::mutex> lock(mutex_);
        file_ << message << '\n';
    }
};
```

This small class composes multiple RAII-managed members:

- the file stream owns the file resource
- the mutex protects shared access
- the lock guard temporarily owns the lock during each call

No manual cleanup code appears in `write`. The design remains robust because each resource manages itself.

19.4.7 It supports modern move-based ownership transfer

Modern C++ strengthened RAII through move semantics. Many resource-owning types should not be copied, because copying could imply duplicate ownership of the same low-level resource. But they should be movable, because ownership sometimes must be transferred.

```
#include <memory>
#include <utility>

std::unique_ptr<int> create_value()
{
    auto p = std::make_unique<int>(99);
    return p;
}

void move_example()
{
    std::unique_ptr<int> a = create_value();
    std::unique_ptr<int> b = std::move(a);
}
```

After the move, ownership transfers from `a` to `b`. This is a natural match for RAII.

19.4.8 It is essential for systems programming on Windows

In Windows programming, many APIs expose raw handles, interface pointers, or manually paired acquire/release functions. Without RAII, code quickly becomes fragile. With RAII wrappers, code becomes safer and more readable.
Typical Windows-oriented RAII targets include:

- `HANDLE`
- file handles
- event handles
- mutex handles
- registry handles
- COM interface pointers
- memory mappings

Whenever an API says “you must call the matching release function,” a C++ programmer should think immediately about an owning wrapper.

19.5 Practical design examples

19.5.1 Example 1: memory buffer owner

```
#include <algorithm>
#include <cstddef>
#include <memory>

class byte_buffer
{
private:
    std::unique_ptr<std::byte[]> data_;
    std::size_t size_{0};

public:
    explicit byte_buffer(std::size_t size)
        : data_(std::make_unique<std::byte[]>(size)), size_(size)
    {
    }

    std::byte* data() noexcept
    {
        return data_.get();
    }

    const std::byte* data() const noexcept
    {
        return data_.get();
    }

    std::size_t size() const noexcept
    {
        return size_;
    }

    void clear() noexcept
    {
        std::fill_n(data_.get(), size_, std::byte{0});
    }
};
```

This class does not expose raw ownership. Internally it uses `std::unique_ptr`, which already provides RAII.

19.5.2 Example 2: scoped Windows handle

```
#include <windows.h>
#include <utility>

class scoped_handle
{
private:
    HANDLE h_{nullptr};
```

```
public:
    scoped_handle() noexcept = default;

    explicit scoped_handle(HANDLE h) noexcept
        : h_(h)
    {
    }

    ~scoped_handle()
    {
        if (h_ && h_ != INVALID_HANDLE_VALUE)
            CloseHandle(h_);
    }

    scoped_handle(const scoped_handle&) = delete;
    scoped_handle& operator=(const scoped_handle&) = delete;

    scoped_handle(scoped_handle&& other) noexcept
        : h_(std::exchange(other.h_, nullptr))
    {
    }

    scoped_handle& operator=(scoped_handle&& other) noexcept
    {
        if (this != &other)
        {
            if (h_ && h_ != INVALID_HANDLE_VALUE)
                CloseHandle(h_);
            h_ = std::exchange(other.h_, nullptr);
        }
        return *this;
    }

    HANDLE get() const noexcept
    {
        return h_;
    }
};
```

The class answers the ownership question unambiguously: one object, one handle owner.

19.5.3 Example 3: scoped lock usage in business code

```
#include <mutex>
#include <string>
#include <vector>

class thread_safe_log
{
private:
    std::mutex mutex_;
```

```
    std::vector<std::string> entries_;

public:
    void add(std::string entry)
    {
        std::lock_guard<std::mutex> lock(mutex_);
        entries_.push_back(std::move(entry));
    }

    std::size_t size() const
    {
        // In real code, mutable mutex or another design may be used here.
        return entries_.size();
    }
};
```

The important design detail is that the function cannot forget to unlock the mutex.

19.5.4 Example 4: transaction-style guard

RAII is not limited to operating system objects. It can model logical resources as well.

```
#include <iostream>

class transaction_guard
{
private:
    bool committed_{false};

public:
    transaction_guard()
    {
        std::cout << "begin transaction\n";
    }

    ~transaction_guard()
    {
        if (!committed_)
            std::cout << "rollback transaction\n";
    }

    void commit()
    {
        std::cout << "commit transaction\n";
        committed_ = true;
    }
};
```

This pattern is common in database and state-management code. Scope exit decides the final action safely.

19.6 Guidelines for writing RAII classes

19.6.1 Acquire in the constructor

If the class is an owner, let construction establish the resource. If acquisition fails, report failure immediately, often by throwing an exception. This prevents half-valid objects from spreading through the program.

19.6.2 Release in the destructor

The destructor should perform the matching cleanup. In general, destructors should not throw. Cleanup code should be reliable and simple.

19.6.3 Delete copying when ownership is exclusive

If the resource cannot be duplicated safely, copy constructor and copy assignment should usually be deleted.

```
my_owner(const my_owner&) = delete;
my_owner& operator=(const my_owner&) = delete;
```

19.6.4 Support moves when transfer makes sense

Move support is often ideal for resource owners because it allows transfer without duplicate ownership.

19.6.5 Provide observers carefully

Functions like `get()` are useful for interoperating with C or Win32 APIs, but they should not weaken ownership rules. Exposing a raw handle or raw pointer for observation is different from transferring responsibility.

19.6.6 Keep invariants simple

A good RAII type often maintains a small invariant such as:

- valid owned resource
- or empty moved-from state

Simple invariants lead to reliable code.

19.7 Common beginner mistakes

19.7.1 Mistake 1: using raw owning pointers

Owning raw pointers make it unclear who must delete the object.

19.7.2 Mistake 2: calling lock and unlock manually

Manual locking is error-prone under exceptions and early returns.

19.7.3 Mistake 3: separating acquire and release across distant code

The farther acquisition and release are separated, the harder it becomes to reason about correctness.

19.7.4 Mistake 4: forgetting that non-memory objects are also resources

Files, handles, sockets, and locks require just as much discipline as heap memory.

19.7.5 Mistake 5: writing move operations incorrectly

A moved-from owner must be left in a safe empty state. Otherwise destruction may double-release the resource.

Summary

Resource management is one of the defining strengths of C++. A resource is any entity that must be acquired and later released correctly. Memory, files, Windows handles, and locks are all examples. The language solves the cleanup problem through deterministic destruction, and modern C++ turns that capability into a design discipline through RAII.

RAII means that ownership is represented by an object whose constructor acquires the resource and whose destructor releases it. This single idea makes exception-safe code practical, reduces leaks, prevents many cleanup errors, and provides a uniform way to manage both library and operating system resources.

For a professional C++ programmer, RAII is not optional knowledge. It is one of the most fundamental mental models in the language. Once understood well, it becomes the natural way to design safe classes, use the standard library correctly, and write robust Windows software that remains clean even as complexity grows.

Copy and Move Semantics

20.1 Copy semantics

Copy semantics means creating or assigning an object by duplicating the value of another object while preserving the source object. In ordinary value-oriented programming, copying is the natural behavior that programmers expect. If one integer is copied to another, the source remains unchanged. Modern C++ extends this same intuitive model to class types.
For many classes, copy semantics means that each subobject is copied. For simple value types, this behavior is usually correct and efficient enough. For resource-owning types, however, copying must be designed carefully. If a class owns a file handle, dynamic memory block, mutex wrapper, or any non-shareable resource, a naive shallow copy may duplicate only the raw handle or raw pointer value rather than duplicating the owned resource. That creates serious bugs such as double deletion, two objects pretending to own the same resource, or use-after-free.
This is why copy semantics is deeply connected to resource management. A class that owns something must decide whether copying is:

- fully supported, with a real deep copy
- disabled, because ownership is unique
- replaced by shared ownership through another design

For a beginner, the important idea is this: copying is not merely copying bits. Copying must preserve the logical meaning of the type.

20.1.1 Default copy behavior

If a class does not explicitly define copy operations, the compiler may generate them. Conceptually, the generated copy constructor and copy assignment operator copy base-class subobjects and non-static data members.
For many types built only from value-like members, this is exactly what you want.

```
#include <string>

struct Person
{
    std::string name;
    int age;
};

int main()
```

```
{
    Person a{"Ayman", 40};
    Person b = a;   // copy construction
    Person c;
    c = a;          // copy assignment
}
```

This is usually correct because `std::string` already manages its own resources safely and supports copying properly.

20.1.2 Why naive copying can be dangerous

Now consider a class that owns raw dynamic memory.

```
class BadBuffer
{
private:
    int* data_;
    std::size_t size_;

public:
    explicit BadBuffer(std::size_t n)
        : data_(new int[n]), size_(n)
    {
    }

    ~BadBuffer()
    {
        delete[] data_;
    }
};
```

This class allocates memory and deletes it, but it does not define copy operations. If copying happens, the compiler-generated copy constructor copies the pointer value, not the array itself. Then two objects point to the same memory, and both destructors try to delete it.

```
int main()
{
    BadBuffer a(10);
    BadBuffer b = a; // dangerous shallow copy
}
```

This is one of the classic motivations for the rule of three, rule of five, and modern RAII design.

20.1.3 A correct deep-copy design

If copying makes sense for the type, then the class must perform a true deep copy.

```
#include <algorithm>
#include <cstddef>

class Buffer
{
```

```
private:
    int* data_;
    std::size_t size_;

public:
    explicit Buffer(std::size_t n)
        : data_(new int[n]{}), size_(n)
    {
    }

    ~Buffer()
    {
        delete[] data_;
    }

    Buffer(const Buffer& other)
        : data_(new int[other.size_]), size_(other.size_)
    {
        std::copy(other.data_, other.data_ + size_, data_);
    }

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

        int* new_data = new int[other.size_];
        std::copy(other.data_, other.data_ + other.size_, new_data);

        delete[] data_;
        data_ = new_data;
        size_ = other.size_;
        return *this;
    }

    std::size_t size() const noexcept
    {
        return size_;
    }

    int& operator[](std::size_t index) noexcept
    {
        return data_[index];
    }

    const int& operator[](std::size_t index) const noexcept
    {
        return data_[index];
    }
};
```

This class preserves copy semantics correctly:

- each copied object gets its own memory
- modifying one object does not affect the other
- destruction is safe

20.1.4 Copy constructor vs copy assignment

These two operations are related but different.

- The **copy constructor** creates a new object from an existing object.
- The **copy assignment operator** assigns to an already existing object from another object.

```
Buffer a(5);
Buffer b = a; // copy constructor

Buffer c(8);
c = a;        // copy assignment
```

The difference matters because assignment must often clean up old owned resources before replacing them.

20.2 Move semantics

Move semantics is one of the defining features of modern C++. It allows resource ownership to be transferred from one object to another instead of duplicated. This is especially valuable for expensive resources such as heap memory, containers, file handles, strings, sockets, and other RAII-managed objects.
The key observation is simple: sometimes the source object is temporary, or the programmer explicitly states that the source object can be treated as expiring. In that case, copying its expensive internal state is unnecessary. It is often better to transfer the resource representation directly.
For example, if an object owns a large dynamic array, copying requires a new allocation and full element copy. Moving can often just transfer the pointer and size, then leave the source in an empty valid state.

20.2.1 Why move semantics matters

Move semantics improves performance and design quality because it allows:

- efficient return-by-value
- efficient container reallocation
- efficient transfer of ownership
- safer replacement of raw-pointer ownership patterns

A moved-from object must remain valid, but its exact value may be changed. In good designs, the moved-from state is usually empty or equivalent to a default-constructed state.

20.2.2 A simple conceptual example

Suppose an object owns a heap block:

```
class SimpleBuffer
{
private:
    int* data_;
    std::size_t size_;
};
```

A copy operation duplicates the array. A move operation can instead transfer:

- `data\_`
- `size\_`

from the source to the destination, then set the source to:

- `nullptr`
- `0`

That is the essence of moving: steal the representation, then leave the source safe to destroy.

20.3 lvalues and rvalues

To understand copy and move semantics, you must understand value categories, especially lvalues and rvalues.

20.3.1 What is an lvalue

An lvalue is an expression that identifies an object with a persistent location that your program can access. Named variables are typical lvalues.

```
int x = 10;
x = 20;
```

Here, `x` is an lvalue. It refers to a real object with identity and storage.
Examples of lvalues include:

- named variables
- array elements
- dereferenced pointers
- functions returning lvalue references

20.3.2 What is an rvalue

An rvalue is a value that is typically temporary or not intended to be referred to long-term. In modern terminology, rvalues include prvalues and xvalues, but for learning move semantics, the practical distinction is usually this:

- an lvalue usually has stable identity in the current program state
- an rvalue is often temporary or treated as expiring

Examples:

```
int x = 10;
int y = x + 5;   // x + 5 is an rvalue
int z = 42;      // 42 is an rvalue
```

20.3.3 Why the distinction matters

Copying is traditionally associated with lvalues because the source is expected to remain usable and unchanged.
Moving is associated with rvalues because temporary or explicitly expiring objects can safely give up their internals.
That is why C++ introduced rvalue references, written with `&&`. They allow overloads and constructors to distinguish movable sources from ordinary named objects.

20.3.4 A simple overload demonstration

```
#include <iostream>

void process(const std::string& s)
{
    std::cout << "lvalue or const reference overload\n";
}

void process(std::string&& s)
{
    std::cout << "rvalue reference overload\n";
}

int main()
{
    std::string name = "Modern C++";

    process(name);                  // lvalue overload
    process(std::string("Temp"));   // rvalue overload
}
```

The named variable `name` is an lvalue even though its type is `std::string`. The temporary `std::string("Temp")` is an rvalue.

20.3.5 Important rule: a named object is an lvalue

This is one of the most important beginner rules in move semantics.
Even if a variable has type `T&&`, once it has a name, the expression that names it is an lvalue.

```
void f(std::string&& s)
{
    // s is a named variable, so the expression s is an lvalue here
}
```

That is one of the main reasons `std::move` exists.

20.4 `std::move`

`std::move` does not move anything by itself. This point is extremely important.
Its job is to convert its argument into an rvalue expression, more precisely an xvalue, so that move-enabled overloads become eligible. It is essentially a cast that says:

> You may treat this object as an expiring value.

If the type actually has a move constructor or move assignment operator, then a real move may happen. If not, the program may fall back to copying.

20.4.1 Basic use of `std::move`

```
#include <string>
#include <utility>

int main()
{
    std::string a = "Resource-rich string";
    std::string b = std::move(a);
}
```

Here, `a` is a named variable, so it is an lvalue. `std::move(a)` casts it to an rvalue expression, enabling `std::string` to move from it. After the move, `a` remains valid, but its value is unspecified in the ordinary logical sense of the moved-from state. You may destroy it, assign to it, or reinitialize it, but you should not rely on its previous contents.

20.4.2 Why `std::move` is needed inside move operations

Consider a move constructor:

```
class Example
{
private:
    std::string name_;

public:
    Example(std::string&& n)
        : name_(n)
    {
    }
};
```

This does *not* move from `n`. Because `n` is named, it is an lvalue expression inside the constructor body and member initializer context. So the code attempts copying.
The correct version is:

```
#include <utility>

class Example
{
private:
    std::string name_;

public:
    Example(std::string&& n)
        : name_(std::move(n))
    {
    }
};
```

Now `name\_` is constructed by moving from `n`.

20.4.3 Do not overuse `std::move`

You should not blindly call `std::move` everywhere. If you move from an object that you still need in its original logical value, the code becomes incorrect.

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

int main()
{
    std::string s = "Important text";
    std::string t = std::move(s);

    // s is valid, but its content should not be relied on as before
    std::cout << "t = " << t << '\n';

    s = "Reused safely";
    std::cout << "s = " << s << '\n';
}
```

A good practical rule is:

- move only when you are intentionally transferring resources
- do not read semantic meaning from the old value after the move
- it is safe to destroy, reassign, or clear a moved-from object

20.5 Move constructor and assignment

Move operations are the special member functions that enable transfer of resources.

20.5.1 Move constructor

A move constructor constructs a new object from an rvalue of the same type.
Its usual form is:

```
T(T&& other) noexcept;
```

Its task is to take ownership of the source object's resources and leave the source in a valid state.

20.5.2 Move assignment operator

A move assignment operator assigns from an rvalue to an already existing object.
Its usual form is:

```
T& operator=(T&& other) noexcept;
```

Its task is usually:

- release the current object's old resource
- steal the source resource
- leave the source valid
- return `*this`

20.5.3 A complete move-enabled class

The following example shows copy constructor, copy assignment, move constructor, move assignment, and destructor together.
This is a classical educational example for a unique resource-owning type.

```
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <utility>

class Buffer
{
private:
    int* data_;
    std::size_t size_;

public:
    explicit Buffer(std::size_t n = 0)
        : data_(n ? new int[n]{} : nullptr), size_(n)
    {
    }

    ~Buffer()
    {
        delete[] data_;
    }
```

```
Buffer(const Buffer& other)
    : data_(other.size_ ? new int[other.size_] : nullptr),
      size_(other.size_)
{
    std::copy(other.data_, other.data_ + size_, data_);
}

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

    int* new_data = other.size_ ? new int[other.size_] : nullptr;
    std::copy(other.data_, other.data_ + other.size_, new_data);

    delete[] data_;
    data_ = new_data;
    size_ = other.size_;
    return *this;
}

Buffer(Buffer&& other) noexcept
    : data_(other.data_), size_(other.size_)
{
    other.data_ = nullptr;
    other.size_ = 0;
}

Buffer& operator=(Buffer&& other) noexcept
{
    if (this == &other)
        return *this;

    delete[] data_;

    data_ = other.data_;
    size_ = other.size_;

    other.data_ = nullptr;
    other.size_ = 0;
    return *this;
}

std::size_t size() const noexcept
{
    return size_;
}

int& operator[](std::size_t index) noexcept
{
    return data_[index];
```

```
    }

    const int& operator[](std::size_t index) const noexcept
    {
        return data_[index];
    }
};
```

20.5.4 How the move constructor works

```
Buffer(Buffer&& other) noexcept
    : data_(other.data_), size_(other.size_)
{
    other.data_ = nullptr;
    other.size_ = 0;
}
```

The steps are:

1. copy the raw representation from `other` into the new object
2. reset `other` so its destructor is harmless

No new allocation occurs. That is why moving can be much cheaper than copying.

20.5.5 How the move assignment works

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

    delete[] data_;

    data_ = other.data_;
    size_ = other.size_;

    other.data_ = nullptr;
    other.size_ = 0;
    return *this;
}
```

This operation must first clean up the current resource, then transfer ownership from `other`.

20.5.6 Why `noexcept` matters

Move operations are often marked `noexcept`. This is not merely style. Standard library containers can prefer moving during reallocation when move construction is known not to throw. A non-`noexcept` move constructor may cause a container to copy instead in situations where stronger exception safety is required.
That is why modern guidance strongly favors:

```
T(T&&) noexcept;
T& operator=(T&&) noexcept;
```

when that guarantee is correct.

20.5.7 Move-only types

Some types should not be copyable at all because they represent unique ownership.
A classic example is `std::unique_ptr`. It can move, but it cannot copy.

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

int main()
{
    std::unique_ptr<int> p1 = std::make_unique<int>(42);

    // std::unique_ptr<int> p2 = p1; // error: copy is not allowed

    std::unique_ptr<int> p2 = std::move(p1);

    if (!p1)
        std::cout << "p1 is now empty\n";

    std::cout << *p2 << '\n';
}
```

This is one of the most important real-world examples of move semantics in modern C++.

20.6 When to copy vs move

Choosing between copy and move is not only a performance question. It is a semantic design question.

20.6.1 Copy when the source must remain unchanged in meaning

Use copying when both source and destination must independently preserve the same logical value.
Examples:

- duplicating a configuration object
- passing a small value type by value when duplication is natural
- preserving a string or vector while also making another copy

```
#include <string>

int main()
{
    std::string original = "Compiler";
    std::string copy = original; // both keep the same text
}
```

20.6.2 Move when ownership or expensive state is being transferred

Use moving when the source object is temporary or no longer needs to preserve its old resource ownership.
Examples:

- returning a large object by value
- pushing a temporary into a container
- transferring a `std::unique_ptr`
- reusing a local object only after reassigning it

```
#include <string>
#include <utility>
#include <vector>

int main()
{
    std::vector<std::string> names;
    std::string value = "Ayman";

    names.push_back(std::move(value));
}
```

After this, `value` is valid but has been moved from.

20.6.3 Copying is often clearer when values are small or cheap

Not every use of move semantics improves code. For small, cheap-to-copy, or trivially copyable types, copying is often clearer and just as good.

```
int a = 10;
int b = a; // simple copy, exactly right
```

Using `std::move` on such types usually adds no meaningful benefit and may reduce clarity.

20.6.4 Move is especially important for resource owners

Move semantics shines most when the type owns something expensive or unique:

- dynamic memory
- strings with large buffers
- vectors and other containers
- file wrappers
- sockets
- handles
- smart pointers

20.6.5 Function parameters: practical guidance

A useful beginner-friendly rule set is:

- Use `const T\&` when the function only reads a possibly expensive object.
- Use `T\&` when the function modifies an existing object.
- Use `T` by value when the function will make its own copy or may move from the argument internally.
- Use `T\&\&` mainly when explicitly designing move-aware or forwarding interfaces.

Example of pass-by-value followed by move into a member:

```
#include <string>
#include <utility>

class Book
{
private:
    std::string title_;

public:
    explicit Book(std::string title)
        : title_(std::move(title))
    {
    }
};
```

This design is elegant because:

- lvalue callers cause one copy into the parameter, then one move into the member
- rvalue callers can move into the parameter, then move into the member

It often simplifies overload sets.

20.7 Copy elision and return values

Modern C++ often constructs returned objects directly in their destination, avoiding both copy and move in many common cases. This is one reason return-by-value is now idiomatic and efficient.

```
#include <string>

std::string make_name()
{
    return std::string("Modern C++");
}
```

In modern compilers, this is typically highly efficient. You should not automatically avoid returning large objects by value out of fear of copying. Move semantics and copy elision changed the performance model of C++ significantly.

20.8 A practical Windows-oriented example

The following example uses `std::vector` and `std::string`, both of which are move-aware and fully supported on Windows with modern compilers such as MSVC.

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

struct FileRecord
{
    std::string file_name;
    std::vector<char> data;

    FileRecord(std::string name, std::vector<char> bytes)
        : file_name(std::move(name)), data(std::move(bytes))
    {
    }
};

FileRecord load_demo_record()
{
    std::string name = "report.bin";
    std::vector<char> bytes(1024, 'A');

    return FileRecord(std::move(name), std::move(bytes));
}

int main()
{
    FileRecord rec = load_demo_record();

    std::cout << rec.file_name << '\n';
    std::cout << "size = " << rec.data.size() << '\n';
}
```

This demonstrates normal modern style:

- use standard-library RAII types
- rely on move-aware constructors
- return objects by value
- avoid manual memory management

20.9 Common mistakes

20.9.1 Mistake 1: believing `std::move` moves by itself

It only casts to an rvalue expression. Actual moving depends on available move operations.

20.9.2 Mistake 2: using a moved-from object as if nothing happened

A moved-from object is valid, but its previous logical value should not be relied upon.

20.9.3 Mistake 3: forgetting that named rvalue references are lvalues

Inside a move constructor or move-aware function, a named parameter must usually be passed onward with `std::move`.

20.9.4 Mistake 4: implementing copy for a unique-ownership type

Some types should not be copied. Forcing copy semantics on such a type often creates ownership bugs.

20.9.5 Mistake 5: forgetting `noexcept` on move operations

Standard containers may benefit more from move operations when they are declared `noexcept`.

20.9.6 Mistake 6: writing move operations that do not leave the source valid

A moved-from object must still be safely destructible and assignable.

20.10 Design guidance for beginners

20.10.1 Prefer the rule of zero when possible

If your class can be built entirely from types that already manage their own resources correctly, then let those members handle copy and move semantics for you.

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

struct Project
{
    std::string name;
    std::vector<std::string> files;
};
```

This class usually needs no custom destructor, copy constructor, move constructor, or assignment operators.

20.10.2 Use smart pointers instead of raw owning pointers

Move semantics becomes far safer and clearer when ownership is represented with standard library types such as `std::unique_ptr`.

20.10.3 Write custom copy and move only when the class directly manages a resource

If your type owns raw memory, raw handles, or other low-level resources, then define the special member functions carefully or redesign around safer building blocks.

Summary

Copy semantics preserves the source and duplicates the logical value of an object. Move semantics transfers resources from an object that is temporary or explicitly treated as expiring. The distinction depends on value categories, especially lvalues and rvalues.
An lvalue usually names a stable object. An rvalue usually represents a temporary or expiring value. `std::move` allows a programmer to treat a named object as movable by casting it to an rvalue expression.
Move constructor and move assignment operator make resource transfer efficient and safe, especially for RAII types and standard library containers. Good move operations leave the source in a valid state and are often marked `noexcept`.
In professional modern C++ design, the question is not merely whether copying or moving is faster. The deeper question is what ownership and value semantics the type should represent. Once that is clear, the correct design often follows naturally:

- copy when two independent values should exist
- move when ownership or expensive state should be transferred
- disable copy when uniqueness is essential
- prefer the rule of zero whenever standard library types can manage everything safely

Smart Pointers

21.1 Why not `new/delete`

In early C++ programming, dynamic memory was managed manually using `new` and `delete`. While this approach provides full control, it is also one of the most common sources of bugs in real-world systems.
The fundamental problems with raw memory management are:

- **Memory leaks**: forgetting to call `delete` leads to unreleased memory
- **Dangling pointers**: accessing memory after it has been deleted
- **Double deletion**: deleting the same pointer more than once
- **Exception unsafety**: early exits or exceptions may skip cleanup

Manual pairing of `new` and `delete` requires discipline across all execution paths. As systems grow in complexity, this becomes error-prone and difficult to maintain.
Smart pointers solve this by applying RAII. They are objects that wrap raw pointers and automatically release the owned resource when they go out of scope.

```
int* p = new int(42);
// many operations...
delete p; // must not be forgotten
```

Modern C++ strongly prefers:

```
#include <memory>

auto p = std::make_unique<int>(42);
// automatic cleanup
```

This eliminates the need for explicit `delete` and ensures exception safety.

21.2 `std::unique_ptr`

`std::unique_ptr` represents **exclusive ownership**. Only one `unique_ptr` can own a resource at a time.

21.2.1 Key properties

- Cannot be copied
- Can be moved
- Automatically deletes the resource when destroyed

21.2.2 Basic usage

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

int main()
{
    auto ptr = std::make_unique<int>(100);
    std::cout << *ptr << '\n';
}
```

21.2.3 Move semantics with `unique_ptr`

```
#include <memory>
#include <utility>

int main()
{
    auto p1 = std::make_unique<int>(10);
    auto p2 = std::move(p1); // ownership transfer
}
```

After the move:

- `p2` owns the resource
- `p1` becomes empty

21.2.4 Custom deleter

```
#include <memory>
#include <windows.h>

struct HandleDeleter
{
    void operator()(HANDLE h) const
    {
        if (h) CloseHandle(h);
    }
};

int main()
{
    std::unique_ptr<void, HandleDeleter> handle(
```

```
        CreateEvent(nullptr, FALSE, FALSE, nullptr)
    );
}
```

This demonstrates how `unique_ptr` integrates with Windows APIs.

21.2.5 When to use

- Default smart pointer choice
- Resource has a single owner
- No shared lifetime required

21.3 `std::shared_ptr`

`std::shared_ptr` represents **shared ownership**. Multiple smart pointers can refer to the same object. Internally, it uses reference counting.

21.3.1 Key properties

- Copyable
- Maintains a reference count
- Object is destroyed when the last owner is destroyed

21.3.2 Basic usage

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

int main()
{
    auto p1 = std::make_shared<int>(42);
    auto p2 = p1;

    std::cout << *p1 << '\n';
}
```

21.3.3 Reference counting behavior

```
#include <memory>

int main()
{
    auto p1 = std::make_shared<int>(5);

    {
        auto p2 = p1; // count = 2
```

```
    } // p2 destroyed, count = 1

} // p1 destroyed, count = 0 -> object deleted
```

21.3.4 When to use

- Multiple parts of the program share ownership
- Lifetime cannot be determined by a single owner

21.3.5 Cost considerations

- Additional memory for control block
- Atomic reference counting (thread-safe overhead)

21.4 `std::weak_ptr`

`std::weak_ptr` represents a **non-owning reference** to an object managed by `std::shared_ptr`. It does not increase the reference count.

21.4.1 Key properties

- Does not affect object lifetime
- Must be converted to `shared_ptr` using `lock()`
- Can detect expired objects

21.4.2 Basic usage

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

int main()
{
    std::shared_ptr<int> sp = std::make_shared<int>(10);
    std::weak_ptr<int> wp = sp;

    if (auto locked = wp.lock())
    {
        std::cout << *locked << '\n';
    }
}
```

If the object is already destroyed:

```
sp.reset();

if (auto locked = wp.lock())
```

```
{
    // not executed
}
```

21.4.3 When to use

- Observing a shared object without extending its lifetime
- Breaking cyclic references

21.5 Ownership models

Smart pointers express ownership directly in the type system. This is one of their most important design contributions.

21.5.1 Exclusive ownership

- Represented by `std::unique_ptr`
- Only one owner exists
- Ownership can be transferred

21.5.2 Shared ownership

- Represented by `std::shared_ptr`
- Multiple owners
- Lifetime ends when last owner is destroyed

21.5.3 Non-owning reference

- Represented by `std::weak_ptr` or raw pointer/reference
- Does not control lifetime
- Used for observation only

21.5.4 Design clarity

Smart pointers encode ownership intent explicitly. This removes ambiguity present in raw pointer APIs.

```
std::unique_ptr<File> open_file(); // caller owns

File* get_file(); // unclear ownership
```

21.6 Cycles and pitfalls

21.6.1 Circular reference problem

A major pitfall of `std::shared_ptr` is cyclic ownership.

```
#include <memory>

struct Node
{
    std::shared_ptr<Node> next;
};

int main()
{
    auto a = std::make_shared<Node>();
    auto b = std::make_shared<Node>();

    a->next = b;
    b->next = a; // cycle
}
```

Here:

- reference count never reaches zero
- memory is never freed

This is a logical memory leak.

21.6.2 Breaking cycles with `weak_ptr`

```
struct Node
{
    std::weak_ptr<Node> next;
};
```

Using `weak_ptr` removes ownership from one side and allows proper cleanup.

21.6.3 Other pitfalls

- Overusing `shared_ptr` when `unique_ptr` is sufficient
- Creating multiple `shared_ptr` from the same raw pointer
- Forgetting that `weak_ptr` must be checked before use
- Assuming smart pointers solve all memory issues

21.7 Best practices

21.7.1 Prefer `std::unique_ptr` by default

It has:

- zero overhead compared to raw pointer
- clear ownership
- no reference counting

21.7.2 Use `std::make_unique` and `std::make_shared`

These functions:

- avoid manual `new`
- improve exception safety

```
auto p1 = std::make_unique<int>(5);
auto p2 = std::make_shared<int>(5);
```

21.7.3 Avoid raw owning pointers

Raw pointers should typically be:

- non-owning observers
- used for interoperability only

21.7.4 Use `std::shared_ptr` only when necessary

Shared ownership has:

- runtime cost
- complexity

Use it only when multiple owners are required.

21.7.5 Use `std::weak_ptr` to break cycles

Whenever a graph or bidirectional relationship exists, consider weak references.

21.7.6 Follow RAII consistently

Smart pointers are part of a larger philosophy:

- resources must have clear ownership
- lifetime must be automatic
- cleanup must be deterministic

Summary

Smart pointers are one of the most important tools in modern C++. They eliminate manual memory management errors and encode ownership semantics directly in the type system.

- `std::unique_ptr`: exclusive ownership, lightweight, default choice
- `std::shared_ptr`: shared ownership with reference counting
- `std::weak_ptr`: non-owning reference to shared objects

The key design principle is not just automatic deletion, but expressing ownership clearly. This leads to safer, more maintainable, and more professional C++ code aligned with RAII and modern best practices.

Error Handling

22.1 Types of errors

Error handling in C++ must address different categories of failures. Understanding these categories is essential before choosing the appropriate mechanism.

22.1.1 Compile-time errors

These errors are detected by the compiler and prevent the program from building.

```
int main()
{
    int x = "text"; // type error
}
```

They are the safest kind of error because they are caught early and cannot reach runtime.

22.1.2 Runtime errors

These occur during program execution:

- file not found
- invalid input
- division by zero
- resource exhaustion

```
#include <iostream>

int main()
{
    int a = 10;
    int b = 0;
    std::cout << a / b; // runtime error
}
```

22.1.3 Logic errors

These are bugs in program logic. The program compiles and runs but produces incorrect results.

```
int sum(int a, int b)
{
    return a - b; // logical error
}
```

22.1.4 Recoverable vs unrecoverable errors

- **Recoverable**: file not found, temporary network failure
- **Unrecoverable**: corrupted memory, violated invariants

This classification strongly influences whether exceptions or alternative mechanisms should be used.

22.2 Exceptions

C++ provides a structured mechanism for handling runtime errors using exceptions.
An exception represents an abnormal condition that disrupts normal control flow. It allows error propagation without cluttering every function with error-checking code.

22.2.1 Basic concept

- `throw`: signal an error
- `try`: define a protected block
- `catch`: handle the error

22.2.2 Throwing an exception

```
#include <stdexcept>

void divide(int a, int b)
{
    if (b == 0)
        throw std::runtime_error("division by zero");
}
```

22.2.3 Standard exception hierarchy

C++ provides standard exception types:

- `std::exception`
- `std::runtime_error`
- `std::logic_error`

- `std::invalid_argument`
- `std::out_of_range`

22.3 `try/catch/throw`

22.3.1 Basic usage

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

int main()
{
    try
    {
        throw std::runtime_error("error occurred");
    }
    catch (const std::runtime_error& e)
    {
        std::cout << e.what() << '\n';
    }
}
```

22.3.2 Multiple catch blocks

```
try
{
    // code
}
catch (const std::invalid_argument& e)
{
}
catch (const std::runtime_error& e)
{
}
catch (...)
{
    // catch-all
}
```

22.3.3 Stack unwinding and RAII

When an exception is thrown:

- stack unwinding begins
- destructors of all automatic objects are called

```
#include <memory>
#include <stdexcept>
```

```
void f()
{
    auto ptr = std::make_unique<int>(10);
    throw std::runtime_error("failure");
}
```

The memory is still released because `unique_ptr` follows RAII.

22.3.4 Rethrowing exceptions

```
try
{
    // code
}
catch (...)
{
    // logging
    throw; // rethrow
}
```

22.4 When to use exceptions

Exceptions are suitable when:

- errors are rare and exceptional
- failure prevents normal continuation
- the caller should decide how to handle the error

Examples:

- file open failure
- invalid user input
- resource acquisition failure

Exceptions should be avoided when:

- failure is expected and frequent
- performance-critical inner loops
- simple control flow decisions

22.4.1 Example: file handling

```
#include <fstream>
#include <stdexcept>

void read_file()
{
    std::ifstream file("data.txt");
    if (!file)
        throw std::runtime_error("cannot open file");
}
```

22.5 `std::optional`

`std::optional` represents a value that may or may not be present.

It is useful when absence of a value is a normal and expected outcome.

22.5.1 Basic usage

```
#include <optional>

std::optional<int> find_value(bool found)
{
    if (found)
        return 42;
    return std::nullopt;
}
```

22.5.2 Checking for value

```
auto result = find_value(false);

if (result)
{
    int value = *result;
}
```

22.5.3 Using value_or

```
int value = result.value_or(0);
```

22.5.4 When to use optional

- absence is not an error
- simple success/failure without details

22.6 `std::expected`

`std::expected` (introduced in C++23) represents either a value or an error.

It is a more expressive alternative to `std::optional` when error information is required.

22.6.1 Basic usage

```
#include <expected>
#include <string>

std::expected<int, std::string> parse_number(const std::string& s)
{
    if (s == "42")
        return 42;
    return std::unexpected("invalid input");
}
```

22.6.2 Handling expected

```
auto result = parse_number("10");

if (result)
{
    int value = *result;
}
else
{
    auto error = result.error();
}
```

22.6.3 Advantages over exceptions

- explicit error handling
- no hidden control flow
- no stack unwinding overhead

22.6.4 When to use expected

- errors are part of normal flow
- need to return error details
- performance-sensitive code

22.7 Writing safe error-handling code

22.7.1 Prefer RAII for cleanup

Never rely on manual cleanup after errors.

```
#include <memory>

void safe()
{
    auto ptr = std::make_unique<int>(5);
}
```

22.7.2 Use exceptions for exceptional failures

- use standard exception types
- avoid throwing primitive types

22.7.3 Avoid resource leaks

Always use RAII wrappers:

- smart pointers
- containers
- lock guards

22.7.4 Keep functions exception-safe

Three levels of exception safety:

- **basic guarantee**: no leaks, valid state
- **strong guarantee**: operation is atomic
- **no-throw guarantee**: never throws

22.7.5 Do not overuse exceptions

Exceptions are powerful but should not replace normal control flow.

22.7.6 Prefer value-based error handling when appropriate

- use `std::optional` for absence
- use `std::expected` for rich errors

22.7.7 Example combining techniques

```
#include <expected>
#include <fstream>
#include <string>

std::expected<std::string, std::string> read_file(const std::string& path)
{
    std::ifstream file(path);
    if (!file)
        return std::unexpected("file open failed");

    std::string content;
    file >> content;
    return content;
}
```

Summary

Error handling in modern C++ provides multiple tools:

- Exceptions for exceptional failures
- `std::optional` for optional values
- `std::expected` for value-or-error results

The key principles are:

- distinguish between error types
- use RAII to ensure cleanup
- choose the correct abstraction for the situation
- keep ownership and lifetime clear

A professional C++ programmer does not rely on a single technique, but selects the most appropriate mechanism based on semantics, performance, and clarity.

Part VII

Templates and Generic Programming

Introduction to Templates

23.1 Function templates

Function templates are one of the most important foundations of modern C++. A function template allows one function definition to work with many different types, as long as the operations used inside the template are valid for those types. Instead of writing separate functions for `int`, `double`, `std::string`, or user-defined types, a single function template can express the common algorithm once and let the compiler generate the needed concrete functions.
This idea is central to generic programming. The programmer describes behavior in terms of requirements on types, not in terms of one fixed type only.

23.1.1 Basic syntax

A function template begins with the `template` keyword and a template parameter list.

```
template <typename T>
T maximum(const T& a, const T& b)
{
    return (a < b) ? b : a;
}
```

In this example:

- `T` is a template type parameter
- the compiler creates a concrete function when the template is used with a specific type
- the expression `a < b` must be valid for the chosen type

23.1.2 Using a function template

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

template <typename T>
T maximum(const T& a, const T& b)
{
    return (a < b) ? b : a;
}

int main()
```

```
{
    std::cout << maximum(10, 20) << '\n';
    std::cout << maximum(3.14, 2.71) << '\n';
    std::cout << maximum(std::string("Alpha"), std::string("Beta")) << '\n';
}
```

The compiler generates different specializations of the function for each suitable type used in the program.

23.1.3 Why function templates are powerful

Function templates remove duplication and improve consistency. Without templates, the following kind of repetition would be common:

```
int maximum_int(int a, int b)
{
    return (a < b) ? b : a;
}

double maximum_double(double a, double b)
{
    return (a < b) ? b : a;
}
```

Templates let the algorithm be written once, reducing maintenance cost and the chance of inconsistent behavior between versions.

23.1.4 Multiple template parameters

A function template can have more than one template parameter.

```
#include <iostream>

template <typename T1, typename T2>
auto add(T1 a, T2 b)
{
    return a + b;
}

int main()
{
    std::cout << add(10, 2.5) << '\n';
}
```

This template accepts two possibly different types and uses `auto` return type deduction to determine the result type.

23.1.5 Explicit template arguments

Often the compiler can deduce template arguments automatically, but you may also provide them explicitly.

```
#include <iostream>

template <typename T>
```

```
T square(T value)
{
    return value * value;
}

int main()
{
    std::cout << square<int>(5) << '\n';
    std::cout << square<double>(2.5) << '\n';
}
```

Explicit template arguments are useful when deduction is impossible, ambiguous, or when the programmer wants to force a specific type.

23.1.6 Function templates and overloads

Function templates can coexist with ordinary functions and with other overloaded templates.

```
#include <iostream>

void print(int value)
{
    std::cout << "non-template int: " << value << '\n';
}

template <typename T>
void print(const T& value)
{
    std::cout << "template: " << value << '\n';
}

int main()
{
    print(42);
    print(3.14);
}
```

In overload resolution, the compiler considers both template and non-template candidates. If template argument deduction succeeds, instantiated function template overloads participate in overload resolution together with non-template functions.

23.1.7 A practical example for Windows-oriented development

Function templates are useful for reusable utility code that works across many value types.

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

template <typename T>
void print_container(const std::vector<T>& values)
{
    for (const auto& value : values)
    {
```

```
        std::cout << value << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> a{1, 2, 3, 4};
    std::vector<double> b{1.5, 2.5, 3.5};

    print_container(a);
    print_container(b);
}
```

This style is highly valuable in real applications because it avoids rewriting container-related helper functions for every element type.

23.2 Class templates

A class template defines a family of related classes. Instead of representing one concrete class, it describes a class pattern parameterized by one or more template arguments.
This is one of the most important concepts in modern C++ because much of the standard library is based on class templates. Examples include:

- `std::vector<T>`
- `std::array<T, N>`
- `std::unique_ptr<T>`
- `std::optional<T>`
- `std::pair<T1, T2>`

23.2.1 Basic syntax

```
template <typename T>
class Box
{
private:
    T value_;

public:
    explicit Box(const T& value)
        : value_(value)
    {
    }

    const T& get() const
    {
```

```
        return value_;
    }
};
```

This template describes a family of classes. For example:

- `Box<int>`
- `Box<double>`
- `Box<std::string>`

are different concrete classes generated from the same template.

23.2.2 Using a class template

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

template <typename T>
class Box
{
private:
    T value_;

public:
    explicit Box(const T& value)
        : value_(value)
    {
    }

    const T& get() const
    {
        return value_;
    }
};

int main()
{
    Box<int> a(100);
    Box<std::string> b("Modern C++");

    std::cout << a.get() << '\n';
    std::cout << b.get() << '\n';
}
```

23.2.3 Class templates with multiple parameters

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

template <typename K, typename V>
```

```
class KeyValue
{
private:
    K key_;
    V value_;

public:
    KeyValue(const K& key, const V& value)
        : key_(key), value_(value)
    {
    }

    const K& key() const
    {
        return key_;
    }

    const V& value() const
    {
        return value_;
    }
};

int main()
{
    KeyValue<int, std::string> item(1, "Compiler");

    std::cout << item.key() << ": " << item.value() << '\n';
}
```

This pattern is common in generic data structures and framework code.

23.2.4 Member functions of class templates

Member functions of class templates may be defined inside the class body or outside it. When defined outside, they are themselves written in template form.

```
#include <iostream>

template <typename T>
class Holder
{
private:
    T value_;

public:
    explicit Holder(const T& value);
    void show() const;
};

template <typename T>
Holder<T>::Holder(const T& value)
```

```
    : value_(value)
{
}

template <typename T>
void Holder<T>::show() const
{
    std::cout << value_ << '\n';
}

int main()
{
    Holder<int> h(50);
    h.show();
}
```

23.2.5 Non-type template parameters

Templates are not limited to types. A template parameter can also represent a constant value.

```
#include <iostream>

template <typename T, int Size>
class FixedBuffer
{
private:
    T data_[Size]{};

public:
    constexpr int size() const
    {
        return Size;
    }

    T& operator[](int index)
    {
        return data_[index];
    }

    const T& operator[](int index) const
    {
        return data_[index];
    }
};

int main()
{
    FixedBuffer<int, 4> buffer;
    buffer[0] = 10;
    buffer[1] = 20;

    std::cout << buffer.size() << '\n';
```

```
    std::cout << buffer[0] << ' ' << buffer[1] << '\n';
}
```

This style is common in compile-time fixed-size abstractions such as arrays, lookup tables, and policy-driven code.

23.2.6 Class templates as the basis of the standard library

A large portion of modern C++ becomes easier to understand once class templates are understood properly. For example:

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

int main()
{
    std::vector<std::string> names;
    names.push_back("Ayman");
    names.push_back("C++");

    for (const auto& name : names)
    {
        std::cout << name << '\n';
    }
}
```

Here, `std::vector<std::string>` is a class template instantiation. The standard library container logic is written once and then reused for many element types.

23.3 Type deduction

Type deduction is one of the most useful aspects of templates. It allows the compiler to infer template arguments from how a function template is called or, in some modern cases, how a class template object is initialized.
This feature makes generic code more natural to use and reduces unnecessary verbosity.

23.3.1 Function template argument deduction

When calling a function template, the compiler tries to determine the template arguments from the function arguments.

```
#include <iostream>

template <typename T>
T minimum(const T& a, const T& b)
{
    return (a < b) ? a : b;
}

int main()
{
    std::cout << minimum(7, 3) << '\n';
    std::cout << minimum(2.5, 9.1) << '\n';
}
```

In this example:

- for `minimum(7, 3)`, the compiler deduces `T = int`
- for `minimum(2.5, 9.1)`, the compiler deduces `T = double`

This makes templates convenient because callers often do not need to write explicit template arguments.

23.3.2 When deduction fails

Deduction can fail if there is not enough information or if the argument types do not match the template pattern.

```
template <typename T>
T minimum(const T& a, const T& b)
{
    return (a < b) ? a : b;
}

int main()
{
    // minimum(10, 2.5); // may fail because T cannot be deduced as one single type
}
```

In such cases, the programmer may:

- provide explicit template arguments
- convert one argument to match the other
- redesign the template to support mixed types

23.3.3 Explicitly guiding deduction

```
#include <iostream>

template <typename T>
T minimum(const T& a, const T& b)
{
    return (a < b) ? a : b;
}

int main()
{
    std::cout << minimum<double>(10, 2.5) << '\n';
}
```

Now both arguments are treated within the chosen type.

23.3.4 Type deduction with forwarding references

A function parameter of the form `T&&` in a template can participate in special deduction rules. This is one of the most important modern uses of template deduction.

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

template <typename T>
void inspect(T&& value)
{
    std::cout << "template called\n";
}

int main()
{
    int x = 10;

    inspect(x);   // lvalue argument
    inspect(20);  // rvalue argument
}
```

In this pattern, deduction behaves differently depending on whether the argument is an lvalue or rvalue. This mechanism is a key building block for perfect forwarding and modern generic library design.

23.3.5 Class template argument deduction

Modern C++ also supports class template argument deduction in many cases. This means the compiler can deduce class template arguments from constructor arguments.

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

int main()
{
    std::pair p(10, std::string("Compiler"));

    std::cout << p.first << '\n';
    std::cout << p.second << '\n';
}
```

Here the compiler deduces the type as approximately equivalent to:

```
std::pair<int, std::string>
```

This feature improves readability and reduces repetition in many common cases.

23.3.6 Why deduction matters

Type deduction brings three major advantages:

- less repetition in code
- easier use of generic abstractions
- fewer opportunities for mismatched manually written types

However, a programmer must still understand what is being deduced. Good template programming depends not only on letting the compiler infer types, but also on understanding the rules well enough to predict and guide the result.

23.4 Benefits of generic programming

Generic programming is a programming style in which algorithms and data structures are written in a way that is independent of specific concrete types, while still remaining type-safe and efficient.
Templates are the core language mechanism that makes this possible in C++.

23.4.1 Code reuse without sacrificing type safety

One of the greatest benefits of generic programming is that it allows a single implementation to serve many types without resorting to untyped programming or unsafe casts.
For example, compare a template-based approach with a weak raw-style approach:

```
template <typename T>
void swap_values(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

This version is type-safe. The compiler checks whether the operations required by the algorithm are valid for the chosen type.

23.4.2 Zero-overhead abstraction

Templates are a major example of zero-overhead abstraction in C++. The abstraction is expressed in source code, but the compiler generates concrete code for the actual types used.
This means generic code can often be as efficient as hand-written type-specific code, while being much more reusable.

23.4.3 Consistency of algorithms

When one template implementation is reused across many types, correctness improvements benefit all those types. This reduces the risk of having different versions of the same algorithm behave differently.

23.4.4 Foundation of the standard library

The standard library is one of the best demonstrations of the power of generic programming.
Containers, iterators, algorithms, smart pointers, tuples, optional values, expected values, and many utilities are template-based.
Once templates are understood, a huge portion of the language ecosystem becomes more understandable.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> values{5, 2, 9, 1, 3};

    std::sort(values.begin(), values.end());

    for (int value : values)
    {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

The generic algorithm `std::sort` works with many iterator and element types, not just integers.

23.4.5 Expressing intent at a higher level

Generic programming encourages thinking in terms of capabilities and interfaces rather than one fixed concrete representation. For example, instead of asking:

> How do I sort a vector of integers?

the generic view becomes:

> How do I sort a range of elements that can be ordered?

This style leads to stronger abstractions and more flexible program design.

23.4.6 Better maintainability in large systems

In large projects, duplicated implementations become expensive to maintain. Templates reduce repetition and centralize logic, making systems easier to evolve.
For example, a reusable template utility can be applied throughout an application, including on Windows desktop tools, backend services, data-processing components, and system utilities, all without rewriting the core algorithm for each data type.

23.4.7 Compile-time checking

Generic code is still checked at compile time. If a type does not satisfy the operations needed by a template, the compiler reports it. This helps catch many errors early.

```cpp
#include <iostream>

template <typename T>
T maximum(const T& a, const T& b)
{
    return (a < b) ? b : a;
```

```
}

struct NoCompare
{
    int value;
};

int main()
{
    // NoCompare a{1}, b{2};
    // auto result = maximum(a, b); // compile-time error: operator< not available
}
```

This is an important strength of C++ generic programming: abstraction does not mean giving up compile-time correctness.

23.5 Practical design examples

23.5.1 Example 1: generic maximum function

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

template <typename T>
T maximum(const T& a, const T& b)
{
    return (a < b) ? b : a;
}

int main()
{
    std::cout << maximum(10, 20) << '\n';
    std::cout << maximum(3.14, 2.71) << '\n';
    std::cout << maximum(std::string("A"), std::string("B")) << '\n';
}
```

23.5.2 Example 2: generic holder class

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

template <typename T>
class Holder
{
private:
    T value_;

public:
    explicit Holder(const T& value)
        : value_(value)
    {
    }
```

```
    const T& get() const
    {
        return value_;
    }
};

int main()
{
    Holder<int> a(100);
    Holder<std::string> b("Templates");

    std::cout << a.get() << '\n';
    std::cout << b.get() << '\n';
}
```

23.5.3 Example 3: compile-time sized buffer

```
#include <iostream>

template <typename T, int N>
class StaticArray
{
private:
    T data_[N]{};

public:
    constexpr int size() const
    {
        return N;
    }

    T& operator[](int index)
    {
        return data_[index];
    }

    const T& operator[](int index) const
    {
        return data_[index];
    }
};

int main()
{
    StaticArray<int, 5> values;
    values[0] = 11;
    values[1] = 22;

    std::cout << values.size() << '\n';
    std::cout << values[0] << ' ' << values[1] << '\n';
}
```

23.5.4 Example 4: template deduction in real use

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

template <typename T>
void display_first(const std::vector<T>& items)
{
    if (!items.empty())
    {
        std::cout << items.front() << '\n';
    }
}

int main()
{
    std::vector<int> numbers{10, 20, 30};
    std::vector<double> values{1.5, 2.5, 3.5};

    display_first(numbers);
    display_first(values);
}
```

23.6 Common beginner mistakes

23.6.1 Mistake 1: assuming templates are macros

Templates are not textual substitution like the preprocessor. They are a typed compile-time mechanism integrated into the language.

23.6.2 Mistake 2: forgetting that operations must exist for the chosen type

A template only works if the used expressions are valid for the provided type. If the template uses <, assignment, or construction, those operations must be available.

23.6.3 Mistake 3: confusing function templates with class templates

Function templates often deduce their arguments automatically. Class templates traditionally require explicit arguments, although class template argument deduction can reduce that requirement in many modern cases.

23.6.4 Mistake 4: writing too many separate overloads instead of one good template

Beginners often duplicate nearly identical code instead of recognizing a generic pattern.

23.6.5 Mistake 5: not understanding deduction failures

When a template call fails, the issue is often not that templates are broken, but that deduction rules or required operations do not match the code.

Summary

Templates are one of the core pillars of modern C++. Function templates define families of related functions, while class templates define families of related classes. Type deduction allows the compiler to infer template arguments in many situations, which makes generic code easier and more natural to use.
The real power of templates appears in generic programming. By expressing algorithms and data structures in terms of capabilities rather than one fixed type, C++ achieves strong code reuse, compile-time type safety, and high performance.
A solid understanding of function templates, class templates, and type deduction is essential because these ideas appear everywhere in modern C++:

- standard library containers
- algorithms
- smart pointers
- optional and expected types
- modern utility classes
- advanced metaprogramming and concepts

Once templates are understood clearly, the rest of generic programming becomes much more approachable.

Type Traits

24.1 What are type traits

Type traits are compile-time utilities from the standard header `<type_traits>` that let a C++ program ask questions about types or transform types without creating runtime overhead.
They are among the most important tools in generic programming because templates often need to behave differently depending on the properties of a type. A template may need to know whether a type is integral, whether two types are identical, whether a type is a reference, or what the plain value form of a type is after removing qualifiers and references.
Type traits are usually grouped into three broad categories:

- **Type predicates**: traits that answer yes or no questions about a type
- **Type relations**: traits that compare two or more types
- **Type transformations**: traits that produce a modified type

Examples:

- `std::is_integral<T>` is a type predicate
- `std::is_same<T, U>` is a type relation
- `std::remove_reference<T>` and `std::decay<T>` are type transformations

A major advantage of type traits is that they operate entirely at compile time. This makes them ideal for template constraints, overload selection, conditional compilation paths, and building generic components that remain efficient.

24.2 `std::is_same`

`std::is_same` is a binary type trait used to test whether two types are exactly the same.
Its basic form is:

```
std::is_same<T, U>::value
```

If `T` and `U` are exactly the same type, the value is `true`. Otherwise it is `false`.

24.2.1 Basic examples

```
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << std::is_same<int, int>::value << '\n';
    std::cout << std::is_same<int, double>::value << '\n';
}
```

Expected behavior:

- the first expression evaluates to true
- the second expression evaluates to false

24.2.2 Exact type comparison

`std::is_same` performs an exact comparison. That means even small differences matter:

- `int` and `const int` are not the same
- `int` and `int\&` are not the same
- `int` and `int&&` are not the same

```
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << std::is_same<int, const int>::value << '\n';
    std::cout << std::is_same<int, int&>::value << '\n';
    std::cout << std::is_same<int, int&&>::value << '\n';
}
```

All three are false because the types are not identical.

24.2.3 Using `std::is_same_v`

Modern C++ provides the variable template form:

```
std::is_same_v<T, U>
```

This is often easier to read.

```
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << std::is_same_v<long, long> << '\n';
    std::cout << std::is_same_v<long, unsigned long> << '\n';
}
```

24.2.4 Why it is useful

`std::is_same` is useful in template code when behavior should depend on exact type identity.
Typical use cases:

- verifying that a deduced type matches an expected type
- checking transformations during template metaprogramming
- selecting code paths for one exact type

24.2.5 Practical example

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

template <typename T>
void describe_type()
{
    if constexpr (std::is_same_v<T, int>)
    {
        std::cout << "T is exactly int\n";
    }
    else if constexpr (std::is_same_v<T, std::string>)
    {
        std::cout << "T is exactly std::string\n";
    }
    else
    {
        std::cout << "T is another type\n";
    }
}

int main()
{
    describe_type<int>();
    describe_type<std::string>();
    describe_type<double>();
}
```

This technique is simple and highly useful for teaching and debugging templates.

24.3 `std::is_integral`

`std::is_integral` tests whether a type is an integral type.
Its basic form is:

```
std::is_integral<T>::value
```

This trait is true for integral types such as:

- `bool`
- `char`
- `signed char`
- `unsigned char`
- `short`
- `unsigned short`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `long long`
- `unsigned long long`
- and their cv-qualified versions

24.3.1 Basic examples

```
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << std::is_integral<int>::value << '\n';
    std::cout << std::is_integral<const unsigned int>::value << '\n';
    std::cout << std::is_integral<double>::value << '\n';
    std::cout << std::is_integral<int*>::value << '\n';
}
```

The first two evaluate to true, while the last two evaluate to false.

24.3.2 Using `std::is_integral_v`

```
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << std::is_integral_v<long long> << '\n';
    std::cout << std::is_integral_v<float> << '\n';
}
```

24.3.3 Why it matters in templates

Many generic algorithms should behave differently for integral types than for floating-point or class types.
Examples:

- choosing integer-specific formatting
- enabling overloads only for integral types
- validating that a template argument is suitable for bitwise operations

24.3.4 Practical example with `if constexpr`

```
#include <type_traits>
#include <iostream>

template <typename T>
void process_value(T value)
{
    if constexpr (std::is_integral_v<T>)
    {
        std::cout << "Integral value: " << value << '\n';
    }
    else
    {
        std::cout << "Non-integral value\n";
    }
}

int main()
{
    process_value(42);
    process_value(3.14);
}
```

24.3.5 Using `std::is_integral` for constraints

Before concepts became widely used, `std::enable_if` and traits such as `std::is_integral` were often used to restrict templates.

```
#include <type_traits>
#include <iostream>

template <typename T>
std::enable_if_t<std::is_integral_v<T>, void>
print_binary_hint(T value)
{
    std::cout << "This function accepts only integral types: "
              << value << '\n';
}

int main()
{
```

```
    print_binary_hint(7);
    // print_binary_hint(3.5); // rejected by the type system
}
```

This remains an important pattern to understand because older codebases and many libraries still use it.

24.4 `std::remove_reference`

`std::remove_reference` is a transformation trait that removes lvalue and rvalue reference qualifiers from a type.
Its basic form is:

```
typename std::remove_reference<T>::type
```

If `T` is:

- `int\&`, the result is `int`
- `int&&`, the result is `int`
- `int`, the result is still `int`

24.4.1 Basic examples

```
#include <type_traits>
#include <iostream>

int main()
{
    std::cout << std::is_same_v<std::remove_reference<int&>::type, int> << '\n';
    std::cout << std::is_same_v<std::remove_reference<int&&>::type, int> << '\n';
    std::cout << std::is_same_v<std::remove_reference<int>::type, int> << '\n';
}
```

All three comparisons evaluate to true.

24.4.2 Using `std::remove_reference_t`

Modern C++ provides the alias:

```
std::remove_reference_t<T>
```

This is shorter and more readable.

```
#include <type_traits>

static_assert(std::is_same_v<std::remove_reference_t<int&>, int>);
static_assert(std::is_same_v<std::remove_reference_t<int&&>, int>);
```

24.4.3 Why it is important

Reference removal is extremely important in template programming because deduced types often carry reference qualifiers. For example, in forwarding references and perfect forwarding, the deduced type may become:

- `T = int\&` for lvalue arguments
- `T = int` for rvalue arguments

If a template needs the underlying object type without reference qualifiers, `std::remove_reference` is a core tool.

24.4.4 Practical example

```
#include <type_traits>
#include <iostream>

template <typename T>
void inspect()
{
    using BaseType = std::remove_reference_t<T>;

    if constexpr (std::is_same_v<BaseType, int>)
    {
        std::cout << "Underlying type is int\n";
    }
    else
    {
        std::cout << "Underlying type is not int\n";
    }
}

int main()
{
    inspect<int>();
    inspect<int&>();
    inspect<int&&>();
}
```

All three instantiations print that the underlying type is `int`.

24.5 `std::decay`

`std::decay` is one of the most useful transformation traits in generic programming. It produces the type that results when a type is passed by value.
This is a very practical definition:

> `std::decay<T>` transforms a type into the form typically used for pass-by-value semantics.

It performs several important adjustments:

- removes references

- removes top-level `const` and `volatile`
- converts arrays to pointers
- converts function types to function pointers

24.5.1 Basic examples

```cpp
#include <type_traits>

static_assert(std::is_same_v<std::decay_t<int>, int>);
static_assert(std::is_same_v<std::decay_t<const int>, int>);
static_assert(std::is_same_v<std::decay_t<int&>, int>);
static_assert(std::is_same_v<std::decay_t<int&&>, int>);
```

24.5.2 Array decay

One of the most important features of `std::decay` is that arrays become pointers.

```cpp
#include <type_traits>

static_assert(std::is_same_v<std::decay_t<int[5]>, int*>);
```

This mirrors the usual behavior of arrays in pass-by-value contexts.

24.5.3 Function decay

Function types decay to function pointers.

```cpp
#include <type_traits>

using Func = int(double);
using Decayed = std::decay_t<Func>;

static_assert(std::is_same_v<Decayed, int(*)(double)>);
```

24.5.4 Why `std::decay` matters

In many templates, the exact original form of a type is too detailed for storage or comparison. A programmer may want the ordinary value form of a type instead.
Examples:

- storing template arguments inside a class
- normalizing deduced types before comparison
- implementing forwarding utilities
- mimicking pass-by-value behavior in metaprogramming

24.5.5 Practical example

```
#include <type_traits>
#include <iostream>

template <typename T>
void show_normalized_type()
{
    using Normalized = std::decay_t<T>;

    if constexpr (std::is_same_v<Normalized, int>)
    {
        std::cout << "Normalized to int\n";
    }
    else if constexpr (std::is_same_v<Normalized, int*>)
    {
        std::cout << "Normalized to int*\n";
    }
    else
    {
        std::cout << "Another normalized type\n";
    }
}

int main()
{
    show_normalized_type<const int>();
    show_normalized_type<int&>();
    show_normalized_type<int[3]>();
}
```

This demonstrates how `std::decay` reduces several related type forms to a normalized result.

24.6 Practical usage

Type traits become truly valuable when used to guide template behavior. This section shows how the four traits from this chapter can work together in realistic code.

24.6.1 Using traits to validate assumptions

```
#include <type_traits>

template <typename T>
void require_integral()
{
    static_assert(std::is_integral_v<T>,
                  "T must be an integral type");
}

int main()
{
```

```
    require_integral<int>();
    // require_integral<double>(); // compile-time error
}
```

This is a simple and powerful pattern. Instead of allowing a misuse to reach runtime, the template rejects incorrect types during compilation.

24.6.2 Comparing transformed types

A common template task is checking whether a type becomes a desired form after removing references.

```
#include <type_traits>
#include <iostream>

template <typename T>
void analyze_reference_removed()
{
    using Plain = std::remove_reference_t<T>;

    if constexpr (std::is_same_v<Plain, int>)
    {
        std::cout << "After removing references, the type is int\n";
    }
    else
    {
        std::cout << "After removing references, the type is not int\n";
    }
}

int main()
{
    analyze_reference_removed<int&>();
    analyze_reference_removed<int&&>();
    analyze_reference_removed<double&>();
}
```

24.6.3 Using `std::decay` for normalized storage types

A template class often needs to store values in normalized form rather than keeping reference-qualified or cv-qualified types directly.

```
#include <type_traits>
#include <utility>
#include <iostream>

template <typename T>
class ValueHolder
{
private:
    std::decay_t<T> value_;

public:
```

```
    explicit ValueHolder(T&& value)
        : value_(std::forward<T>(value))
    {
    }

    const std::decay_t<T>& get() const
    {
        return value_;
    }
};

int main()
{
    int x = 42;
    ValueHolder<int&> a(x);
    ValueHolder<const int> b(100);

    std::cout << a.get() << '\n';
    std::cout << b.get() << '\n';
}
```

Here, `std::decay_t<T>` ensures the stored member is a value-like type rather than a reference type.

24.6.4 Integral-only helper function

```
#include <type_traits>
#include <iostream>

template <typename T>
void print_if_integral(T value)
{
    if constexpr (std::is_integral_v<T>)
    {
        std::cout << "Integral: " << value << '\n';
    }
    else
    {
        std::cout << "Not integral\n";
    }
}

int main()
{
    print_if_integral(10);
    print_if_integral(3.5);
}
```

24.6.5 Checking exact type after normalization

```
#include <type_traits>
#include <iostream>
```

```
template <typename T>
void inspect_type(T&& value)
{
    using Raw = std::remove_reference_t<T>;
    using Normalized = std::decay_t<T>;

    if constexpr (std::is_same_v<Raw, int>)
    {
        std::cout << "Raw without reference is int\n";
    }

    if constexpr (std::is_same_v<Normalized, int>)
    {
        std::cout << "Normalized type is int\n";
    }

    (void)value;
}

int main()
{
    int x = 5;
    inspect_type(x);
    inspect_type(10);
}
```

This style is especially useful when studying template deduction or building generic utilities.

24.7 A Windows-oriented example

The following example is standard modern C++ and works naturally in a Windows environment using MSVC, Visual Studio, or CMake-based projects.

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

template <typename T>
void describe_parameter(T&& value)
{
    using NoRef = std::remove_reference_t<T>;
    using Decayed = std::decay_t<T>;

    std::cout << "is_integral<NoRef>: "
              << std::is_integral_v<NoRef> << '\n';

    std::cout << "NoRef is same as int: "
              << std::is_same_v<NoRef, int> << '\n';

    std::cout << "Decayed is same as std::string: "
              << std::is_same_v<Decayed, std::string> << '\n';
```

```
    (void)value;
}

int main()
{
    int number = 25;
    const std::string text = "Modern C++";

    describe_parameter(number);
    describe_parameter(100);
    describe_parameter(text);
}
```

This example shows how multiple traits can cooperate:

- `std::remove_reference_t` strips references
- `std::decay_t` normalizes for value semantics
- `std::is_same_v` checks exact type identity
- `std::is_integral_v` checks numeric category

24.8 Best practices

24.8.1 Prefer alias and variable template forms

Modern C++ is clearer with:

- `std::remove_reference_t<T>` instead of `typename std::remove_reference<T>::type`
- `std::decay_t<T>` instead of `typename std::decay<T>::type`
- `std::is_same_v<T, U>` instead of `std::is_same<T, U>::value`
- `std::is_integral_v<T>` instead of `std::is_integral<T>::value`

These forms reduce noise and make template code easier to read.

24.8.2 Use `std::remove_reference` when you need only reference stripping

If the problem is only about removing `\&` or `\&\&`, use `std::remove_reference`. Do not use `std::decay` unless you also want cv-removal and array/function decay.

24.8.3 Use `std::decay` when modeling pass-by-value behavior

`std::decay` is excellent when a template needs the ordinary stored value type.

24.8.4 Use `std::is_same` carefully

It checks exact type identity. If you need to ignore references or cv-qualifiers, transform the type first before comparing.

```
#include <type_traits>

static_assert(std::is_same_v<std::remove_reference_t<int&>, int>);
static_assert(std::is_same_v<std::decay_t<const int>, int>);
```

24.8.5 Use traits to improve compile-time diagnostics

Traits are especially valuable when combined with `static_assert` and `if constexpr`. This makes templates safer and easier to understand.

24.9 Common beginner mistakes

24.9.1 Mistake 1: comparing unnormalized types

A beginner may expect:

```
std::is_same_v<int, int&>
```

to be true, but it is false because the types are not identical.

24.9.2 Mistake 2: using `std::decay` when only reference removal was needed

`std::decay` does more than remove references. It also removes top-level cv-qualifiers and converts arrays and functions. That may be too strong for some tasks.

24.9.3 Mistake 3: forgetting alias helpers

Writing:

```
typename std::remove_reference<T>::type
```

works, but modern code is usually cleaner with:

```
std::remove_reference_t<T>
```

24.9.4 Mistake 4: assuming traits operate at runtime

Type traits are compile-time tools. They do not inspect dynamic runtime values.

24.9.5 Mistake 5: overcomplicating code

Traits are powerful, but the clearest design is usually the best one. Use them where they improve correctness, constraints, and readability.

Summary

Type traits are a major part of modern C++ generic programming. They allow templates to classify, compare, and transform types at compile time.
The traits in this chapter each serve a different purpose:

- `std::is_same` checks exact type identity
- `std::is_integral` checks whether a type belongs to the integral category
- `std::remove_reference` strips lvalue and rvalue reference qualifiers
- `std::decay` normalizes a type into its pass-by-value form

Together, these tools help C++ programmers write templates that are:

- safer
- more expressive
- easier to constrain
- easier to debug
- better aligned with real type semantics

A strong understanding of these four traits provides an excellent foundation for more advanced topics such as `std::enable\_if`, SFINAE, forwarding, perfect forwarding, concepts, and constrained templates.

Concepts

25.1 Motivation

Concepts are one of the most important improvements added to modern C++ template programming. Their purpose is to express, directly in the source code, what kinds of types and operations a template requires.
Before concepts, templates were often written in a highly open form. A template parameter such as `typename T` could accept almost any type at first glance, but the template body might later use operations that only make sense for a much smaller set of types. As a result, the real requirements of the template were hidden inside the implementation.
This created several problems:

- template interfaces were harder to read
- template requirements were not stated explicitly
- invalid template arguments often produced long and confusing errors
- library authors had to rely on indirect techniques to restrict usage

Concepts solve this by letting the programmer describe requirements in a clear and declarative way.
Instead of writing a template that only *implicitly* requires a type to support comparison, addition, division, iteration, or construction, a concept-based design can state those requirements directly in the template interface.

25.1.1 The basic idea

A concept is a compile-time predicate that describes whether a type or set of types satisfies certain requirements. It can be used to constrain templates, function overloads, class templates, and other generic code.
A concept does not create a runtime object. It is a compile-time rule used by the compiler when checking template arguments.

25.1.2 Why concepts matter in real code

Concepts improve generic programming in several important ways:

- they make template interfaces easier to understand
- they reject invalid template arguments earlier
- they provide a better vocabulary for generic design
- they improve compiler diagnostics for incorrect uses

- they reduce many old patterns based on complicated trait expressions

Consider a simple template that divides two values:

```
template <typename T>
T divide(T a, T b)
{
    return a / b;
}
```

At first glance, this template appears to accept any type `T`. But that is not true. It only works for types that support the division operator. If a programmer tries to instantiate it with an unsuitable type, the compiler error may point deep into the function body.
With concepts, the interface can say that requirement explicitly.

```
#include <concepts>

template <typename T>
concept Divisible = requires(T a, T b)
{
    a / b;
};

template <Divisible T>
T divide(T a, T b)
{
    return a / b;
}
```

Now the template communicates its requirements much more clearly.

25.1.3 Concepts and the evolution of C++ templates

Templates made generic programming possible. Type traits, `std::enable_if`, and SFINAE made template selection more powerful. Concepts build on that history and offer a cleaner language-level way to express many common constraints.
This makes concepts a major improvement in the transition from “templates that happen to fail if misused” to “templates that state exactly what they need.”

25.2 Concepts vs SFINAE

To understand the value of concepts, it is useful to compare them with SFINAE-based techniques.
SFINAE stands for *Substitution Failure Is Not An Error*. It is a rule of template substitution that allows some invalid template substitutions to be ignored rather than treated as hard errors during overload resolution.
For many years, SFINAE was one of the main tools for constraining templates indirectly. It remains important to understand, because older code and many libraries still use it. However, concepts were introduced to make such code more direct, readable, and diagnosable.

25.2.1 A traditional SFINAE-style example

A common older pattern uses `std::enable_if` and a type trait:

```
#include <type_traits>
#include <iostream>

template <typename T>
std::enable_if_t<std::is_integral_v<T>, void>
print_value(T value)
{
    std::cout << "Integral value: " << value << '\n';
}
```

This works, but the constraint is expressed indirectly through a return type transformation. The programmer must read the trait logic to understand that the function is intended only for integral types.

25.2.2 The same idea with a concept

With concepts, the intent becomes much clearer:

```
#include <concepts>
#include <iostream>

template <std::integral T>
void print_value(T value)
{
    std::cout << "Integral value: " << value << '\n';
}
```

This version is easier to read. The template parameter itself expresses the rule.

25.2.3 Why concepts are generally better than SFINAE for interface constraints

Concepts have several advantages over SFINAE-style restriction:

- the constraint appears where readers expect it
- the code is shorter and easier to understand
- the compiler can report constraint failure more directly
- intent is clearer for maintainers and library users

SFINAE often hides the rule in:

- return types
- default template parameters
- nested trait expressions
- helper aliases

Concepts move the rule into the interface itself.

25.2.4 SFINAE is still relevant

Concepts do not make historical knowledge of SFINAE useless. Many important codebases still use SFINAE, especially code written before C++20 or code designed for broad compiler compatibility.
Also, some advanced metaprogramming patterns still rely on lower-level substitution behavior. But for most ordinary constraint-style template design in modern C++, concepts are the cleaner first choice.

25.2.5 A side-by-side comparison

SFINAE style:

```
#include <type_traits>

template <typename T, typename = std::enable_if_t<std::is_integral_v<T>>>
T add_one(T value)
{
    return value + 1;
}
```

Concept style:

```
#include <concepts>

template <std::integral T>
T add_one(T value)
{
    return value + 1;
}
```

Both forms constrain the template. The second form is usually easier to teach, easier to read, and easier to maintain.

25.3 Writing constraints

Concepts can be written and applied in several ways. This section introduces the most practical forms used in modern C++.

25.3.1 Using standard library concepts

The standard library provides a set of ready-made concepts in the header `<concepts>`. These include concepts such as:

- `std::same_as`
- `std::integral`
- `std::floating_point`
- `std::convertible_to`
- `std::derived_from`

A simple example:

```
#include <concepts>
#include <iostream>

template <std::floating_point T>
T half(T value)
{
    return value / static_cast<T>(2);
}

int main()
{
    std::cout << half(10.0) << '\n';
    std::cout << half(7.5f) << '\n';
}
```

This template only accepts floating-point types.

25.3.2 Defining a custom concept

A custom concept is usually written with the `concept` keyword.

```
#include <concepts>

template <typename T>
concept Incrementable = requires(T x)
{
    ++x;
    x++;
};
```

This concept states that `T` must support both prefix and postfix increment.

25.3.3 Constraining a function template

A concept can constrain a function template directly.

```
#include <concepts>
#include <iostream>

template <typename T>
concept Incrementable = requires(T x)
{
    ++x;
    x++;
};

template <Incrementable T>
void advance_once(T& value)
{
    ++value;
}
```

```
int main()
{
    int x = 10;
    advance_once(x);
    std::cout << x << '\n';
}
```

25.3.4 Using a requires clause

Another common style is the `requires` clause.

```
#include <concepts>

template <typename T>
requires std::integral<T>
T multiply_by_two(T value)
{
    return value * 2;
}
```

This form is useful when constraints become longer or when multiple conditions must be combined.

25.3.5 Combining constraints

Concepts can be combined with logical operators such as `&&` and `||`.

```
#include <concepts>

template <typename T>
requires (std::integral<T> || std::floating_point<T>)
T square(T value)
{
    return value * value;
}
```

This template accepts either integral or floating-point types.

25.3.6 Requires expressions

A requires expression is one of the most powerful parts of concepts. It checks whether certain expressions are valid for given types.

```
#include <concepts>

template <typename T>
concept Addable = requires(T a, T b)
{
    a + b;
};
```

This says that `T` must support the expression `a + b`.
A more refined concept can also constrain result types:

```
#include <concepts>

template <typename T>
concept SelfAddable = requires(T a, T b)
{
    { a + b } -> std::same_as<T>;
};
```

Now the expression `a + b` must not only exist, but must also produce exactly `T`.

25.3.7 Constraining class templates

Concepts also work with class templates.

```
#include <concepts>

template <std::integral T>
class Counter
{
private:
    T value_{};

public:
    void increment()
    {
        ++value_;
    }

    T value() const
    {
        return value_;
    }
};
```

This class template can only be instantiated with integral types.

25.3.8 Abbreviated function templates

Modern C++ also allows a shorter form using constrained `auto` parameters.

```
#include <concepts>
#include <iostream>

void show(std::integral auto value)
{
    std::cout << value << '\n';
}
```

This is concise and useful for small generic functions.

25.4 Improving diagnostics

One of the biggest practical benefits of concepts is improved compiler diagnostics.
Before concepts, when a template was instantiated with an unsuitable type, the error often appeared deep inside the template body. The compiler might produce a long chain of messages about invalid operators, missing members, failed substitutions, or unrelated internal helper types.
Concepts move the failure closer to the interface.

25.4.1 The old problem

Consider a traditional unconstrained template:

```
template <typename T>
T divide(T a, T b)
{
    return a / b;
}
```

If a programmer tries to use a type that has no division operator, the compiler reports a failure from inside the implementation. In large template code, this can become difficult to interpret.

25.4.2 The concept-based improvement

Now consider the constrained version:

```
#include <concepts>

template <typename T>
concept Divisible = requires(T a, T b)
{
    a / b;
};

template <Divisible T>
T divide(T a, T b)
{
    return a / b;
}
```

If a type does not satisfy the concept, the compiler can report that the associated constraints are not satisfied. This is usually much closer to the real design intent of the template. MSVC documents dedicated concept-related diagnostics for unsatisfied associated constraints.

25.4.3 Why this helps beginners and experts

Better diagnostics are valuable for both learners and experienced developers:

- beginners see what requirement was missing
- library users get clearer error messages

- maintainers can understand template misuse more quickly
- template interfaces become more self-documenting

25.4.4 Designing for good diagnostics

To get the best results from concepts:

- write small, meaningful concepts
- use descriptive concept names
- constrain templates at the interface, not deep in the body
- prefer readable requirements over clever metaprogramming tricks

A concept named `Sortable`, `Addable`, or `StreamInsertable` explains more than a dense trait expression.

25.4.5 Example of a descriptive concept

```
#include <concepts>
#include <iostream>

template <typename T>
concept StreamWritable = requires(std::ostream& os, const T& value)
{
    os << value;
};

template <StreamWritable T>
void print_line(const T& value)
{
    std::cout << value << '\n';
}
```

This concept clearly communicates why a type is accepted or rejected.

25.5 Practical examples

This section collects realistic examples that show how concepts can improve ordinary generic code on Windows using modern MSVC and C++20 mode.

25.5.1 Example 1: integral-only helper

```
#include <concepts>
#include <iostream>

template <std::integral T>
T next_value(T value)
{
    return value + 1;
```

```
}

int main()
{
    std::cout << next_value(41) << '\n';
    // std::cout << next_value(3.14) << '\n'; // rejected
}
```

This example shows a direct replacement for many older `std::enable_if` patterns.

25.5.2 Example 2: custom addable concept

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

template <typename T>
concept Addable = requires(T a, T b)
{
    { a + b } -> std::same_as<T>;
};

template <Addable T>
T add_values(const T& a, const T& b)
{
    return a + b;
}

int main()
{
    std::cout << add_values(10, 20) << '\n';
    std::cout << add_values(std::string("Modern "), std::string("C++")) << '\n';
}
```

This concept is simple but useful for teaching expression requirements and return-type constraints.

25.5.3 Example 3: constraining a printing function

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

template <typename T>
concept Printable = requires(std::ostream& os, const T& value)
{
    os << value;
};

template <Printable T>
void print_twice(const T& value)
{
    std::cout << value << " | " << value << '\n';
```

```
}

int main()
{
    print_twice(123);
    print_twice(std::string("Templates"));
}
```

This is a highly practical pattern for utility libraries.

25.5.4 Example 4: concept with a requires clause

```
#include <concepts>
#include <iostream>

template <typename T>
requires std::integral<T> && sizeof(T) >= 4
void process_large_integral(T value)
{
    std::cout << "Accepted value: " << value << '\n';
}

int main()
{
    process_large_integral(100);
}
```

This example shows that concepts and ordinary compile-time conditions can be combined naturally.

25.5.5 Example 5: class template with a concept

```
#include <concepts>
#include <iostream>

template <std::integral T>
class IdGenerator
{
private:
    T current_{0};

public:
    T next()
    {
        return ++current_;
    }
};

int main()
{
    IdGenerator<int> ids;
    std::cout << ids.next() << '\n';
    std::cout << ids.next() << '\n';
```

```
}
```

This is a clean example of constraining a class template to its intended domain.

25.5.6 Example 6: abbreviated function template

```
#include <concepts>
#include <iostream>

void display_number(std::integral auto value)
{
    std::cout << "number = " << value << '\n';
}

int main()
{
    display_number(25);
}
```

This style is compact and expressive for small utilities.

25.5.7 Example 7: a concept for equality comparability

```
#include <concepts>
#include <iostream>

template <typename T>
concept EqualityComparable = requires(const T& a, const T& b)
{
    { a == b } -> std::convertible_to<bool>;
    { a != b } -> std::convertible_to<bool>;
};

template <EqualityComparable T>
bool are_different(const T& a, const T& b)
{
    return a != b;
}

int main()
{
    std::cout << are_different(10, 20) << '\n';
}
```

This kind of concept is common in generic library design.

25.6 Design advice

25.6.1 Prefer standard concepts when they express the requirement well

If the standard library already provides a concept such as `std::integral`, `std::floating_point`, or `std::same_as`, prefer using it rather than creating a custom duplicate.

25.6.2 Write custom concepts for domain meaning

Custom concepts are most valuable when they communicate library-specific or algorithm-specific requirements clearly. Examples:

- `Serializable`
- `Hashable`
- `MatrixLike`
- `LoggerSink`

25.6.3 Keep concepts focused

A concept should usually describe one coherent requirement family. Smaller concepts are easier to reuse and combine.

25.6.4 Use concepts to describe interfaces, not just to restrict arbitrarily

The best concepts explain what a template logically needs, not merely what types the author wanted to permit.

25.7 Common beginner mistakes

25.7.1 Mistake 1: using concepts as if they were runtime checks

Concepts are compile-time constraints. They do not test runtime values.

25.7.2 Mistake 2: writing constraints too late

If the requirement is essential to the interface, express it in the template declaration rather than waiting for a failure in the function body.

25.7.3 Mistake 3: making concepts too complicated too early

Start with small readable concepts. Avoid turning every concept into a dense metaprogramming exercise.

25.7.4 Mistake 4: recreating standard concepts unnecessarily

When `std::integral` or `std::same_as` already expresses the intent, reuse it.

25.7.5 Mistake 5: forgetting that good names improve diagnostics

A concept named `SupportsOperation42` communicates far less than a concept named `Addable` or `Printable`.

Summary

Concepts are a major step forward in modern C++ generic programming. They allow template requirements to be stated directly at the interface level, making code more readable, more maintainable, and easier to diagnose when used incorrectly. Microsoft Learn documents concepts as compile-time constraints that prevent incorrect template instantiation and make template requirements more readable.

Compared with traditional SFINAE-based techniques, concepts usually provide a cleaner and more expressive way to constrain templates. They work naturally with standard concepts, custom concepts, requires clauses, and requires expressions. In practice, they help C++ programmers write generic code that is not only powerful, but also understandable.

For modern C++ development on Windows with MSVC in C++20 mode or later, concepts are one of the most important tools for building safer and more professional template code. MSVC tracks concept support as part of its C++20 language conformance, and Visual Studio tooling also includes diagnostics support for concept-related failures.

Lambda Expressions

26.1 Lambda basics

Lambda expressions are anonymous function objects that can be defined directly at the point of use. They are one of the most important features introduced in modern C++ (since C++11) and significantly extended in later standards (C++14, C++17, C++20, and C++23).
A lambda allows a programmer to write small, inline functions without defining a separate named function or class. Internally, a lambda is translated by the compiler into a unique unnamed class type with an `operator()`.

26.1.1 Basic syntax

The general form of a lambda expression is:

```
[capture](parameters) -> return_type
{
    // body
}
```

A minimal lambda:

```
#include <iostream>

int main()
{
    auto hello = []()
    {
        std::cout << "Hello from lambda\n";
    };

    hello();
}
```

26.1.2 Lambda with parameters and return value

```
#include <iostream>

int main()
{
    auto add = [](int a, int b)
    {
```

```
        return a + b;
    };

    std::cout << add(3, 4) << '\n';
}
```

In most cases, the return type is deduced automatically. An explicit return type can be written if needed:

```
auto divide = [](double a, double b) -> double
{
    return a / b;
};
```

26.1.3 Lambda as a function object

Every lambda has a unique type. It behaves like a class with a call operator.

```
auto square = [](int x)
{
    return x * x;
};
```

This is conceptually similar to:

```
struct Square
{
    int operator()(int x) const
    {
        return x * x;
    }
};
```

26.2 Captures

Captures allow a lambda to access variables from its surrounding scope. This is one of the most powerful features of lambdas.

26.2.1 Capture by value

```
#include <iostream>

int main()
{
    int x = 10;

    auto f = [x]()
    {
        std::cout << x << '\n';
    };

    f();
}
```

The value of `x` is copied into the lambda.

26.2.2 Capture by reference

```
#include <iostream>

int main()
{
    int x = 10;

    auto f = [&x]()
    {
        x += 5;
    };

    f();
    std::cout << x << '\n';
}
```

The lambda modifies the original variable.

26.2.3 Default captures

- `[=]` capture all used variables by value
- `[&]` capture all used variables by reference

```
int a = 1;
int b = 2;

auto sum = [=]()
{
    return a + b;
};
```

26.2.4 Mixed capture

```
int a = 1;
int b = 2;

auto f = [a, &b]()
{
    b += a;
};
```

26.2.5 Mutable lambdas

By default, lambdas that capture by value treat captured variables as const. To modify them, use `mutable`.

```
#include <iostream>

int main()
{
    int x = 10;
```

```
    auto f = [x]() mutable
    {
        x += 5;
        std::cout << x << '\n';
    };

    f();
}
```

26.2.6 Init captures (C++14)

Init capture allows creating new variables inside the lambda capture list.

```
#include <memory>

int main()
{
    auto ptr = std::make_unique<int>(42);

    auto f = [p = std::move(ptr)]()
    {
        // p owns the resource
    };
}
```

This is essential when transferring ownership into a lambda.

26.3 Passing lambdas

Lambdas can be passed to functions, stored in variables, or used as arguments in generic code.

26.3.1 Passing to a template function

```
#include <iostream>

template <typename Func>
void execute(Func f)
{
    f();
}

int main()
{
    execute([]()
    {
        std::cout << "Lambda executed\n";
    });
}
```

Templates are the most efficient way to accept lambdas because they avoid type erasure.

26.3.2 Using `auto` parameters (C++20)

```
#include <iostream>

void call(auto f)
{
    f();
}

int main()
{
    call([]()
    {
        std::cout << "Called\n";
    });
}
```

26.3.3 Using `std::function`

```
#include <functional>
#include <iostream>

void run(std::function<void()> f)
{
    f();
}

int main()
{
    run([]()
    {
        std::cout << "Using std::function\n";
    });
}
```

`std::function` provides type erasure but introduces overhead. Prefer templates when performance matters.

26.4 Usage with algorithms

Lambdas are heavily used with standard library algorithms. This is one of their most practical and common uses.

26.4.1 Using with `std::for_each`

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

int main()
{
    std::vector<int> values{1, 2, 3, 4};
```

```
    std::for_each(values.begin(), values.end(), [](int v)
    {
        std::cout << v << ' ';
    });
}
```

26.4.2 Using with `std::sort`

```
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> values{5, 2, 9, 1};

    std::sort(values.begin(), values.end(), [](int a, int b)
    {
        return a > b;
    });
}
```

26.4.3 Using with `std::find_if`

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

int main()
{
    std::vector<int> values{10, 15, 20, 25};

    auto it = std::find_if(values.begin(), values.end(), [](int v)
    {
        return v > 18;
    });

    if (it != values.end())
    {
        std::cout << *it << '\n';
    }
}
```

26.4.4 Capturing state in algorithms

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

int main()
{
    std::vector<int> values{1, 2, 3, 4};
    int threshold = 2;
```

```
    auto count = std::count_if(values.begin(), values.end(),
        [threshold](int v)
        {
            return v > threshold;
        });

    std::cout << count << '\n';
}
```

26.5 Functional-style programming

Lambda expressions enable a functional programming style within C++.

26.5.1 Key characteristics

- functions are treated as values
- small, composable operations
- reduced need for explicit loops
- emphasis on immutability

26.5.2 Transforming data

```
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> values{1, 2, 3, 4};
    std::vector<int> result(values.size());

    std::transform(values.begin(), values.end(), result.begin(),
        [](int v)
        {
            return v * v;
        });
}
```

26.5.3 Filtering data

```
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> values{1, 2, 3, 4, 5};
    std::vector<int> filtered;
```

```
    std::copy_if(values.begin(), values.end(), std::back_inserter(filtered),
        [](int v)
        {
            return v % 2 == 0;
        });
}
```

26.5.4 Combining operations

```
#include <algorithm>
#include <vector>

int main()
{
    std::vector<int> values{1, 2, 3, 4, 5};
    std::vector<int> result;

    std::copy_if(values.begin(), values.end(), std::back_inserter(result),
        [](int v)
        {
            return v > 2;
        });

    std::transform(result.begin(), result.end(), result.begin(),
        [](int v)
        {
            return v * 2;
        });
}
```

26.5.5 Generic lambdas (C++14)

```
#include <iostream>

int main()
{
    auto add = [](auto a, auto b)
    {
        return a + b;
    };

    std::cout << add(1, 2) << '\n';
    std::cout << add(2.5, 3.5) << '\n';
}
```

Generic lambdas are templates internally and work with many types.

26.5.6 Lambdas and templates together

```
#include <iostream>

template <typename Func, typename T>
```

```
T apply(Func f, T value)
{
    return f(value);
}

int main()
{
    auto square = [](int x) { return x * x; };

    std::cout << apply(square, 5) << '\n';
}
```

This combination is fundamental in modern C++ design.

26.6 Best practices

26.6.1 Keep lambdas small

Lambdas are best used for short, focused operations. If a lambda becomes large, consider a named function.

26.6.2 Capture only what is needed

Avoid capturing everything unnecessarily. Prefer explicit captures.

26.6.3 Prefer value capture for safety

Capturing by value avoids lifetime issues. Use reference capture only when necessary.

26.6.4 Use generic lambdas when flexibility is required

Generic lambdas simplify many template use cases.

26.6.5 Prefer templates over std::function in performance-critical code

`std::function` introduces overhead. Templates allow full optimization.

Summary

Lambda expressions are a core feature of modern C++ that enable concise, expressive, and efficient code.
They provide:

- inline anonymous functions
- access to surrounding variables through captures
- seamless integration with algorithms
- support for functional-style programming

Combined with templates, type traits, and concepts, lambdas form a powerful foundation for modern generic programming. They are widely used in standard library algorithms, asynchronous programming, event handling, and high-performance systems.

A strong understanding of lambdas is essential for writing modern, clean, and professional C++ code.

Part VIII

The Modern Standard Library

Utility Types

27.1 Overview

Modern C++ provides a rich set of utility types in the standard library that help represent common programming patterns such as grouping values, optional values, type-safe unions, and error handling.
These types are defined primarily in the headers:

- `<utility>` for `std::pair`
- `<tuple>` for `std::tuple`
- `<optional>` for `std::optional`
- `<variant>` for `std::variant`
- `<any>` for `std::any`
- `<expected>` for `std::expected`

These abstractions are widely used in modern C++ to improve safety, expressiveness, and clarity.

27.2 `std::pair`

`std::pair` is a simple structure that holds two values, possibly of different types.

27.2.1 Basic usage

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

int main()
{
    std::pair<int, double> p{10, 3.14};

    std::cout << p.first << " " << p.second << '\n';
}
```

27.2.2 Using `std::make_pair`

```
auto p = std::make_pair(42, 2.5);
```

27.2.3 Structured bindings (C++17)

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

int main()
{
    std::pair<int, int> p{1, 2};

    auto [a, b] = p;

    std::cout << a << " " << b << '\n';
}
```

27.2.4 Typical use cases

- returning two values from a function
- representing key-value pairs
- intermediate algorithm results

27.3 `std::tuple`

`std::tuple` generalizes `std::pair` to hold any number of values of different types.

27.3.1 Basic usage

```
#include <tuple>
#include <iostream>

int main()
{
    std::tuple<int, double, const char*> t{1, 3.14, "C++"};

    std::cout << std::get<0>(t) << '\n';
    std::cout << std::get<1>(t) << '\n';
    std::cout << std::get<2>(t) << '\n';
}
```

27.3.2 Using `std::make_tuple`

```
auto t = std::make_tuple(10, 2.5, "text");
```

27.3.3 Structured bindings

```
#include <tuple>
#include <iostream>

int main()
```

```
{
    auto t = std::make_tuple(1, 2.5, "hello");

    auto [a, b, c] = t;

    std::cout << a << " " << b << " " << c << '\n';
}
```

27.3.4 Returning multiple values

```
#include <tuple>

std::tuple<int, int> get_values()
{
    return {1, 2};
}
```

27.4 `std::optional`

`std::optional` represents an object that may or may not contain a value.

27.4.1 Basic usage

```
#include <optional>
#include <iostream>

std::optional<int> find(bool ok)
{
    if (ok)
        return 42;
    return std::nullopt;
}

int main()
{
    auto result = find(true);

    if (result)
    {
        std::cout << *result << '\n';
    }
}
```

27.4.2 Using `value_or`

```
int value = result.value_or(0);
```

27.4.3 Why optional is useful

- avoids sentinel values

- expresses absence explicitly
- safer alternative to raw pointers for optional return values

27.5 `std::variant`

`std::variant` is a type-safe union that can hold one of several types.

27.5.1 Basic usage

```
#include <variant>
#include <iostream>

int main()
{
    std::variant<int, double> v;

    v = 10;
    std::cout << std::get<int>(v) << '\n';

    v = 3.14;
    std::cout << std::get<double>(v) << '\n';
}
```

27.5.2 Using `std::visit`

```
#include <variant>
#include <iostream>

int main()
{
    std::variant<int, double> v = 5;

    std::visit([](auto value)
    {
        std::cout << value << '\n';
    }, v);
}
```

27.5.3 Handling multiple types

```
#include <variant>
#include <iostream>

int main()
{
    std::variant<int, std::string> v = "text";

    std::visit([](auto&& value)
    {
        std::cout << value << '\n';
```

```
    }, v);
}
```

27.5.4 Advantages

- type-safe alternative to unions
- avoids undefined behavior
- enables pattern-like handling via `std::visit`

27.6 `std::any`

`std::any` can hold a value of any type.

27.6.1 Basic usage

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

int main()
{
    std::any a;

    a = 10;
    std::cout << std::any_cast<int>(a) << '\n';

    a = std::string("C++");
    std::cout << std::any_cast<std::string>(a) << '\n';
}
```

27.6.2 Checking type

```
if (a.type() == typeid(int))
{
}
```

27.6.3 Notes

- uses type erasure
- incurs runtime overhead
- should be used when type is not known at compile time

27.7 `std::expected`

`std::expected` (C++23) represents either a value or an error.

27.7.1 Basic usage

```
#include <expected>
#include <string>

std::expected<int, std::string> parse(bool ok)
{
    if (ok)
        return 42;
    return std::unexpected("error");
}
```

27.7.2 Handling expected

```
#include <iostream>

int main()
{
    auto result = parse(true);

    if (result)
    {
        std::cout << *result << '\n';
    }
    else
    {
        std::cout << result.error() << '\n';
    }
}
```

27.7.3 Advantages

- explicit error handling
- avoids exceptions when not desired
- combines value and error in one type

27.8 Comparison and usage guidance

- `std::pair`: simple two-value grouping
- `std::tuple`: multiple heterogeneous values
- `std::optional`: value may be absent
- `std::variant`: one of several types
- `std::any`: completely type-erased storage
- `std::expected`: value or error result

27.9 Best practices

27.9.1 Prefer specific types over generic ones

Use:

- `std::variant` instead of `std::any` when possible
- `std::optional` instead of nullable pointers

27.9.2 Use structured bindings for readability

```cpp
auto [x, y] = std::pair{1, 2};
```

27.9.3 Avoid overusing tuples

If a tuple becomes complex, prefer a struct with named members.

27.9.4 Use expected for error handling

Prefer `std::expected` when errors are part of normal control flow.

Summary

Utility types in modern C++ provide powerful abstractions for common programming patterns.
They improve:

- type safety
- code clarity
- expressiveness
- correctness

Mastering these types is essential for writing modern, maintainable, and professional C++ applications.

Time and Random

28.1 `std::chrono`

The `std::chrono` library provides strongly typed facilities for working with time points, durations, clocks, calendar values, and modern civil-time representations. It is one of the most important parts of the modern C++ standard library because it replaces weak, error-prone time handling based on plain integers, loosely interpreted units, and older C-style APIs with a type-safe and expressive design.
The main ideas in `std::chrono` are:

- a **duration** represents an amount of time
- a **time_point** represents a point on a specific clock's timeline
- a **clock** provides the current time and defines the epoch and tick period
- newer chrono facilities also support **calendar dates**, **time zones**, and **civil time conversions**

A major strength of `std::chrono` is that units are part of the type system. This prevents many classic mistakes such as confusing seconds with milliseconds or mixing unrelated integer values without clear meaning.

28.1.1 Basic duration example

```
#include <chrono>
#include <iostream>

int main()
{
    std::chrono::seconds s(5);
    std::chrono::milliseconds ms(250);

    auto total = s + ms;

    std::cout << std::chrono::duration_cast<std::chrono::milliseconds>(total).count()
              << '\n';
}
```

In this example:

- `seconds` and `milliseconds` are different types
- the library performs unit-aware arithmetic

- the final value can be converted explicitly with `duration\_cast`

28.1.2 Why chrono is safer than raw integers

Without `std::chrono`, code often looks like this:

```
int timeout = 5000;
```

What does `5000` mean?

- milliseconds
- microseconds
- seconds

With chrono, the meaning is explicit:

```
auto timeout = std::chrono::milliseconds(5000);
```

This is clearer, safer, and easier to maintain.

28.1.3 Common chrono headers and types

For this chapter, the most important header is:

```
#include <chrono>
```

Frequently used types include:

- `std::chrono::duration`
- `std::chrono::time\_point`
- `std::chrono::system\_clock`
- `std::chrono::steady\_clock`
- `std::chrono::high\_resolution\_clock`
- `std::chrono::milliseconds`, `seconds`, `minutes`, `hours`

In more modern date/time code, additional chrono facilities include:

- `year`, `month`, `day`
- `year\_month\_day`
- `zoned\_time`
- `utc\_clock`
- `file\_clock`
- `local\_t`

28.2 Clocks and durations

Understanding clocks and durations is the foundation of correct time handling in modern C++.

28.2.1 Durations

A duration represents a measured amount of time. It is not tied to a specific calendar date or moment. It is just a span.
Examples:

- 3 seconds
- 150 milliseconds
- 2 hours

28.2.2 Creating durations

```
#include <chrono>
#include <iostream>

int main()
{
    std::chrono::seconds a(2);
    std::chrono::milliseconds b(500);

    auto c = a + b;

    std::cout << std::chrono::duration_cast<std::chrono::milliseconds>(c).count()
              << '\n';
}
```

28.2.3 Accessing the numeric value

The `count()` member returns the stored numeric representation.

```
#include <chrono>
#include <iostream>

int main()
{
    std::chrono::minutes m(3);
    std::cout << m.count() << '\n';
}
```

28.2.4 Duration casting

When converting between units, use `duration\_cast` where appropriate.

```
#include <chrono>
#include <iostream>
```

```cpp
int main()
{
    std::chrono::milliseconds ms(3500);

    auto s = std::chrono::duration_cast<std::chrono::seconds>(ms);

    std::cout << s.count() << '\n';
}
```

28.2.5 Time points

A `time\_point` represents a moment on the timeline of a particular clock. A time point is meaningful only relative to its clock. For example:

- a `system\_clock::time\_point` represents a wall-clock time
- a `steady\_clock::time\_point` represents a monotonic clock moment for interval measurement

28.2.6 Getting the current time

```cpp
#include <chrono>

int main()
{
    auto now1 = std::chrono::system_clock::now();
    auto now2 = std::chrono::steady_clock::now();
}
```

These time points come from different clocks and are intended for different purposes.

28.2.7 Important standard clocks

The most commonly used clocks are:

- `system\_clock`
- `steady\_clock`
- `high\_resolution\_clock`

28.2.8 `system_clock`

`system\_clock` represents the system wall clock. It corresponds to the current real-world clock time of the machine. It is suitable when you need calendar-related or human-meaningful current time.
Use it for:

- current date and time
- timestamps for logs
- converting to calendar/civil time

However, it is not steady. That means the system time may be adjusted by the operating system, user settings, synchronization services, or daylight-saving-related changes in surrounding conversions.

28.2.9 `steady_clock`

`steady\_clock` is monotonic. Its reported time only moves forward. This makes it ideal for measuring elapsed time and benchmarking intervals.
Use it for:

- timing algorithm execution
- measuring elapsed durations
- timeouts
- retry windows

28.2.10 `high_resolution_clock`

`high\_resolution\_clock` provides the smallest tick period exposed by the implementation. In Microsoft's implementation, it is an alias of `steady\_clock`. Therefore, on MSVC it behaves as a steady monotonic clock suitable for interval timing. This is important for Windows-focused code because many examples online assume `high\_resolution\_clock` is always a distinct special clock, which is not guaranteed by the standard. On MSVC it effectively maps to `steady\_clock`.

28.2.11 Measuring elapsed time correctly

```
#include <chrono>
#include <iostream>
#include <thread>

int main()
{
    auto start = std::chrono::steady_clock::now();

    std::this_thread::sleep_for(std::chrono::milliseconds(150));

    auto end = std::chrono::steady_clock::now();
    auto elapsed = end - start;

    std::cout
        << std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).count()
        << '\n';
}
```

This is the correct pattern for elapsed timing because it uses a steady monotonic clock.

28.2.12 A Windows-oriented timing helper

```
#include <chrono>
#include <iostream>
```

```
template <typename Func>
void measure_ms(Func f)
{
    auto start = std::chrono::steady_clock::now();
    f();
    auto end = std::chrono::steady_clock::now();

    auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
    std::cout << "Elapsed: " << elapsed.count() << " ms\n";
}

int main()
{
    measure_ms([]()
    {
        volatile long long sum = 0;
        for (int i = 0; i < 1000000; ++i)
        {
            sum += i;
        }
    });
}
```

28.2.13 Sleep functions

Time durations integrate naturally with thread waiting functions.

```
#include <chrono>
#include <thread>

int main()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
}
```

This is clearer and safer than passing plain integers to platform-specific APIs.

28.3 Modern time handling

Modern C++ time handling goes beyond measuring elapsed durations. It also includes more expressive civil-time and calendar-oriented facilities.

28.3.1 Wall-clock time vs elapsed time

A key design rule is:

- use `system\_clock` for current real-world time
- use `steady\_clock` for measuring intervals

Confusing these roles is a common beginner error.

28.3.2 Getting the current wall-clock time

```
#include <chrono>
#include <ctime>
#include <iostream>

int main()
{
    auto now = std::chrono::system_clock::now();
    std::time_t t = std::chrono::system_clock::to_time_t(now);

    std::cout << std::ctime(&t);
}
```

This example converts a chrono wall-clock time point to a traditional `time\_t` value for simple display.

28.3.3 Converting from `time_t`

```
#include <chrono>
#include <ctime>
#include <iostream>

int main()
{
    std::time_t t = std::time(nullptr);
    auto tp = std::chrono::system_clock::from_time_t(t);

    auto seconds = std::chrono::time_point_cast<std::chrono::seconds>(tp);
    std::cout << seconds.time_since_epoch().count() << '\n';
}
```

28.3.4 Using calendar types

Modern chrono also supports expressive calendar-oriented types.

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::chrono;

    year_month_day ymd{year{2026}, month{3}, day{20}};

    std::cout << int(ymd.year()) << '-'
              << unsigned(ymd.month()) << '-'
              << unsigned(ymd.day()) << '\n';
}
```

These types are strongly typed and more expressive than manually managing raw integers for year, month, and day.

28.3.5 Building a date from calendar types

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::chrono;

    auto d = year{2026} / month{3} / day{20};

    std::cout << int(d.year()) << ' '
              << unsigned(d.month()) << ' '
              << unsigned(d.day()) << '\n';
}
```

28.3.6 Converting calendar dates to time points

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::chrono;

    sys_days day_point = year{2026}/month{3}/day{20};

    std::cout << day_point.time_since_epoch().count() << '\n';
}
```

This makes calendar-oriented programming much clearer.

28.3.7 UTC, local, and file-related clocks

Modern chrono also introduces additional clock domains such as:

- `utc\_clock`
- `file\_clock`
- local-time tagged time points using `local\_t`

These support more precise modeling of time domains:

- `utc\_clock` represents UTC time and includes leap-second-aware semantics
- `file\_clock` represents file-system related timestamps
- `local\_t` indicates local-time representation without itself encoding a specific time zone

These types are especially useful in modern systems code, file metadata handling, and timezone-aware applications.

28.3.8 Modern timezone-aware thinking

When dealing with human time, a professional design should separate:

- storage of machine time
- calendar interpretation
- local-time presentation
- timezone conversion

Modern chrono provides facilities such as `zoned\_time`, timezone database access types, and supporting calendar conversions for this purpose. On current Microsoft Learn documentation, chrono includes `time\_zone`, `time\_zone\_link`, `tzdb`, `tzdb\_list`, and `zoned\_time`.

28.3.9 Practical logging timestamp example

For many Windows desktop or backend applications, a common practical task is to generate a timestamp.

```
#include <chrono>
#include <ctime>
#include <iomanip>
#include <iostream>

int main()
{
    auto now = std::chrono::system_clock::now();
    std::time_t t = std::chrono::system_clock::to_time_t(now);

    std::tm local_tm{};
#ifdef _WIN32
    localtime_s(&local_tm, &t);
#else
    local_tm = *std::localtime(&t);
#endif

    std::cout << std::put_time(&local_tm, "%Y-%m-%d %H:%M:%S") << '\n';
}
```

This is practical and works well for Windows-oriented console or application logging.

28.4 Random number generation

Modern C++ random number generation is provided by the `<random>` header. It is significantly better than the old C library approach based on `rand()`.
The old `rand()` approach has several weaknesses:

- limited quality
- poor statistical properties in many implementations

- awkward scaling to custom ranges
- global-state style design

Modern C++ instead separates random generation into two concepts:

- **engines**, which generate pseudo-random sequences
- **distributions**, which map engine output into desired statistical forms

This separation is one of the most important design improvements.

28.4.1 Basic engine and distribution example

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

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

    for (int i = 0; i < 5; ++i)
    {
        std::cout << dist(gen) << '\n';
    }
}
```

This example simulates rolling a die.

28.4.2 Why this design is better

The engine and distribution are separate:

- `std::mt19937` produces a pseudo-random sequence
- `std::uniform\_int\_distribution<int>` maps it to a uniform integer range

This is flexible and explicit.

28.4.3 Common engines

Common engines include:

- `std::mt19937`
- `std::mt19937\_64`
- linear congruential engines
- subtract-with-carry engines

For many general applications, `std::mt19937` is a common practical choice.

28.4.4 Seeding the engine

A pseudo-random engine should be seeded. A common approach is to use `std::random\_device`.

```
#include <random>

int main()
{
    std::random_device rd;
    std::mt19937 gen(rd());
}
```

28.4.5 Uniform integer distribution

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

int main()
{
    std::mt19937 gen(std::random_device{}());
    std::uniform_int_distribution<int> dist(100, 999);

    std::cout << dist(gen) << '\n';
}
```

This generates a uniformly distributed integer in the range `[100, 999]`.

28.4.6 Uniform real distribution

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

int main()
{
    std::mt19937 gen(std::random_device{}());
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    std::cout << dist(gen) << '\n';
}
```

This generates a floating-point value in the specified interval.

28.4.7 Normal distribution

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

int main()
{
    std::mt19937 gen(std::random_device{}());
    std::normal_distribution<double> dist(0.0, 1.0);
```

```
    for (int i = 0; i < 5; ++i)
    {
        std::cout << dist(gen) << '\n';
    }
}
```

This is useful for simulations, measurements, and probabilistic modeling.

28.4.8 A reusable integer RNG wrapper

```
#include <random>

class RandomInt
{
private:
    std::mt19937 gen_;

public:
    RandomInt()
        : gen_(std::random_device{}())
    {
    }

    int operator()(int min_value, int max_value)
    {
        std::uniform_int_distribution<int> dist(min_value, max_value);
        return dist(gen_);
    }
};

int main()
{
    RandomInt rnd;
    int value = rnd(1, 10);
    (void)value;
}
```

This is a good practical pattern for application-level code.

28.4.9 Generating random values in containers

```
#include <algorithm>
#include <iostream>
#include <random>
#include <vector>

int main()
{
    std::vector<int> values(10);

    std::mt19937 gen(std::random_device{}());
    std::uniform_int_distribution<int> dist(1, 100);
```

```
    std::generate(values.begin(), values.end(), [&]()
    {
        return dist(gen);
    });

    for (int v : values)
    {
        std::cout << v << ' ';
    }
    std::cout << '\n';
}
```

28.4.10 Shuffling with modern random facilities

The standard library algorithm `std::shuffle` is designed to work with a uniform random number generator.

```
#include <algorithm>
#include <iostream>
#include <random>
#include <vector>

int main()
{
    std::vector<int> values{1, 2, 3, 4, 5};

    std::mt19937 gen(std::random_device{}());
    std::shuffle(values.begin(), values.end(), gen);

    for (int v : values)
    {
        std::cout << v << ' ';
    }
    std::cout << '\n';
}
```

This is the preferred modern technique for random shuffling.

28.5 Best practices

28.5.1 Use the correct clock for the task

A strong practical rule is:

- use `steady\_clock` for elapsed timing
- use `system\_clock` for current wall-clock time

28.5.2 Avoid measuring intervals with `system_clock`

Because the system clock can be adjusted, interval measurement should prefer a steady monotonic clock.

28.5.3 Prefer chrono durations over raw integers

Instead of writing raw timeout integers, use explicit units:

```
auto timeout = std::chrono::milliseconds(500);
```

28.5.4 Use modern chrono date/time types for clarity

When representing civil dates, prefer expressive chrono types such as:

- `year`
- `month`
- `day`
- `year\_month\_day`

28.5.5 Prefer `<random>` over `rand()`

Modern random facilities are more expressive, safer, and more statistically sound for normal application use.

28.5.6 Keep the engine alive when generating many values

Do not reseed the engine before every random number. Usually, create the engine once and reuse it.

28.5.7 Choose the distribution explicitly

Do not scale raw engine output manually when a standard distribution already models the needed behavior.

28.6 Common beginner mistakes

28.6.1 Mistake 1: using `system_clock` for benchmarking

This may give unreliable interval measurements if the system time changes.

28.6.2 Mistake 2: storing time in plain integers without units

This reduces clarity and increases the risk of mixing milliseconds, seconds, and other units incorrectly.

28.6.3 Mistake 3: assuming `high_resolution_clock` is always unique

On MSVC, it is an alias of `steady\_clock`.

28.6.4 Mistake 4: reseeding the random engine too often

This can reduce quality and make behavior less predictable than intended.

28.6.5 Mistake 5: using `rand()` in new code

Modern C++ code should prefer `<random>`.

Summary

Modern C++ provides a powerful and expressive model for time and random number generation.
With `std::chrono`, a programmer gains:

- strong typing for durations and time points
- clear separation between wall-clock time and elapsed timing
- modern calendar and timezone-oriented abstractions
- better safety and readability than raw integer-based time code

With `<random>`, a programmer gains:

- explicit pseudo-random engines
- clear statistical distributions
- better design than legacy C random functions
- reusable and maintainable random generation code

For professional modern C++ on Windows, these libraries are essential. They provide both correctness and clarity, which are especially important in performance measurement, timeouts, logging, simulations, file metadata handling, application scheduling, and any code that depends on reliable time or high-quality random values.

Filesystem

29.1 `std::filesystem`

The `std::filesystem` library provides a standard, portable, and type-safe way to interact with files, directories, and paths. It was introduced in C++17 and significantly improves upon older approaches that relied on platform-specific APIs or manual string manipulation.
The library is defined in the header:

```
#include <filesystem>
```

and is available under the namespace:

```
namespace fs = std::filesystem;
```

29.1.1 Core design goals

- provide a unified interface for file system operations
- abstract platform differences (Windows vs POSIX)
- represent paths as structured objects rather than raw strings
- provide both exception-based and error-code-based APIs

29.1.2 Basic example

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    fs::path p = "example.txt";

    if (fs::exists(p))
    {
        std::cout << "File exists\n";
    }
}
```

29.2 Paths

The `std::filesystem::path` class represents a filesystem path in a structured and portable way.

29.2.1 Creating paths

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    fs::path p1 = "C:\\\\Users\\\\Public\\\\file.txt";
    fs::path p2 = "folder/subfolder/file.txt";
}
```

Paths can be constructed from strings and automatically adapt to the platform.

29.2.2 Combining paths

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    fs::path base = "C:\\\\Data";
    fs::path full = base / "logs" / "app.log";

    std::cout << full.string() << '\n';
}
```

The `/` operator performs path concatenation in a portable way.

29.2.3 Accessing components

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    fs::path p = "C:\\\\Data\\\\file.txt";

    std::cout << p.filename() << '\n';
    std::cout << p.extension() << '\n';
    std::cout << p.parent_path() << '\n';
}
```

29.2.4 Path normalization

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    fs::path p = "C:\\\\Data\\\\..\\\\Logs\\\\file.txt";

    std::cout << p.lexically_normal() << '\n';
}
```

29.2.5 Absolute and relative paths

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    fs::path p = "file.txt";

    std::cout << fs::absolute(p) << '\n';
    std::cout << fs::current_path() << '\n';
}
```

29.3 File operations

The filesystem library provides many operations to query and manipulate files.

29.3.1 Checking file existence

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    if (fs::exists("test.txt"))
    {
    }
}
```

29.3.2 Checking file type

```
#include <filesystem>

namespace fs = std::filesystem;
```

```
int main()
{
    if (fs::is_regular_file("file.txt"))
    {
    }

    if (fs::is_directory("folder"))
    {
    }
}
```

29.3.3 File size

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    auto size = fs::file_size("file.txt");
    std::cout << size << '\n';
}
```

29.3.4 Copying files

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    fs::copy("source.txt", "destination.txt",
             fs::copy_options::overwrite_existing);
}
```

29.3.5 Renaming files

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    fs::rename("old.txt", "new.txt");
}
```

29.3.6 Deleting files

```
#include <filesystem>
```

```
namespace fs = std::filesystem;

int main()
{
    fs::remove("file.txt");
}
```

29.3.7 Last write time

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    auto time = fs::last_write_time("file.txt");
}
```

29.3.8 Permissions

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    fs::permissions("file.txt",
        fs::perms::owner_read,
        fs::perm_options::add);
}
```

29.3.9 Using error codes instead of exceptions

```
#include <filesystem>
#include <system_error>

namespace fs = std::filesystem;

int main()
{
    std::error_code ec;
    fs::remove("file.txt", ec);

    if (ec)
    {
    }
}
```

This is useful for robust systems where exceptions are not desired.

29.4 Directory management

The filesystem library provides powerful tools for working with directories.

29.4.1 Creating directories

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    fs::create_directory("new_folder");
    fs::create_directories("a/b/c");
}
```

29.4.2 Removing directories

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    fs::remove("empty_folder");
    fs::remove_all("folder_with_contents");
}
```

29.4.3 Iterating over directory contents

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
{
    for (const auto& entry : fs::directory_iterator("C:\\\\Data"))
    {
        std::cout << entry.path() << '\n';
    }
}
```

29.4.4 Recursive iteration

```
#include <filesystem>
#include <iostream>

namespace fs = std::filesystem;

int main()
```

```
{
    for (const auto& entry : fs::recursive_directory_iterator("C:\\\\Data"))
    {
        std::cout << entry.path() << '\n';
    }
}
```

29.4.5 Checking directory contents

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    for (const auto& entry : fs::directory_iterator("."))
    {
        if (entry.is_regular_file())
        {
        }
    }
}
```

29.4.6 Changing current directory

```
#include <filesystem>

namespace fs = std::filesystem;

int main()
{
    fs::current_path("C:\\\\Temp");
}
```

29.5 Windows-specific considerations

29.5.1 Path separators

On Windows, both backslash and forward slash may be used. However, internally the library handles normalization automatically.

29.5.2 Wide character support

```
fs::path p = L"C:\\\\Users\\\\Public";
```

Windows uses UTF-16 internally, and `std::filesystem::path` supports wide strings.

29.5.3 Long paths

Modern Windows supports long paths. `std::filesystem` handles them correctly when system settings allow it.

29.6 Best practices

29.6.1 Use `std::filesystem::path` instead of raw strings

Paths should not be manipulated manually using string concatenation.

29.6.2 Prefer portable path composition

Use:

```
path / "subfolder" / "file.txt";
```

instead of manual string building.

29.6.3 Handle errors explicitly in system tools

Use error codes when writing robust tools and services.

29.6.4 Avoid assumptions about path formats

Let the library handle platform-specific details.

29.6.5 Use recursive iteration carefully

Recursive traversal can be expensive and should be controlled.

29.7 Common beginner mistakes

29.7.1 Using string concatenation for paths

This leads to portability issues.

29.7.2 Ignoring error handling

Filesystem operations can fail due to permissions, missing files, or locks.

29.7.3 Assuming all paths exist

Always check with `exists()`.

29.7.4 Confusing files and directories

Always verify type with `is\_regular\_file()` or `is\_directory()`.

Summary

The `std::filesystem` library is a modern, robust, and portable solution for file and directory management in C++.
It provides:

- structured path handling
- safe file operations
- powerful directory traversal
- cross-platform abstraction

For modern C++ development on Windows and other platforms, it replaces older approaches and offers a consistent, expressive, and reliable interface for filesystem interaction.

Text and Unicode

30.1 Encoding fundamentals

Text handling is one of the most misunderstood areas in software development because a *character*, a *code point*, a *code unit*, a *glyph*, and a *byte sequence* are not the same thing. Modern C++ provides several character types and string forms, but a programmer must understand the underlying encoding model to use them correctly.
At the most practical level, text processing requires answering these questions:

- What abstract text is being represented
- What encoding transforms that text into code units
- What concrete storage type is used in memory
- What API or file format expects that representation

30.1.1 Characters, code points, and code units

A modern Unicode-oriented mental model should separate the following concepts:

- A **character** is a human-level text element in ordinary discussion
- A **Unicode code point** is an abstract numeric value such as U+0041 or U+1F600
- A **code unit** is the storage unit used by a particular encoding
- A **glyph** is a visual rendering shape, which may depend on font and context

For example, the same Unicode text can be encoded differently:

- UTF-8 uses 8-bit code units
- UTF-16 uses 16-bit code units
- UTF-32 uses 32-bit code units

This distinction is essential because string length, indexing, file size, and API compatibility all depend on encoding, not only on abstract text.

30.1.2 Character types in modern C++

Modern C++ provides multiple character types, each with a distinct role.

- `char`
- `wchar_t`
- `char8_t`
- `char16_t`
- `char32_t`

30.1.3 `char`

`char` is the traditional narrow character type. It is commonly used with `std::string`. Historically, programs often stored locale-dependent or ASCII-compatible text in `char`-based strings.
In modern code, `char` is still widely used for:

- ordinary byte-oriented text storage
- UTF-8 data in many existing libraries and APIs
- interoperability with classic narrow-string interfaces

However, `char` itself does not guarantee a particular encoding. The meaning depends on the source file encoding, compiler options, literals used, runtime conventions, and external APIs.

30.1.4 `wchar_t`

`wchar_t` is a wide character type. On Windows, it is closely tied to UTF-16-based wide APIs and is commonly used with `std::wstring`. It remains important in Windows programming because many Win32 Unicode APIs use wide strings.

30.1.5 `char8_t`

`char8_t` was added in C++20 as a distinct type for UTF-8 code units. This is an important modernization because it gives UTF-8 text a stronger type identity than plain `char`.

```
#include <iostream>

int main()
{
    const char8_t* text = u8"Hello UTF-8";
    (void)text;
}
```

30.1.6 `char16_t` and `char32_t`

These types represent 16-bit and 32-bit code units respectively and are intended for UTF-16 and UTF-32 oriented text.

```
int main()
{
    const char16_t* t16 = u"Unicode";
    const char32_t* t32 = U"Unicode";
    (void)t16;
    (void)t32;
}
```

30.1.7 String literal prefixes

Modern C++ provides literal prefixes that express character width and intended encoding form.

- ordinary string literal: `"text"`
- wide string literal: `L"text"`
- UTF-8 string literal: `u8"text"`
- UTF-16 string literal: `u"text"`
- UTF-32 string literal: `U"text"`

```
int main()
{
    auto a = "plain narrow";
    auto b = L"wide";
    auto c = u8"utf8";
    auto d = u"utf16";
    auto e = U"utf32";

    (void)a;
    (void)b;
    (void)c;
    (void)d;
    (void)e;
}
```

30.1.8 Source encoding and execution encoding

A C++ program must also consider:

- the **source character set**, which interprets the source file text
- the **execution character set**, which affects ordinary narrow literals

On Windows with MSVC, using UTF-8 source files and enabling UTF-8-related compiler options is an important practical step in modern multilingual development.

30.1.9 A practical Windows compilation note

For Windows development with MSVC, a common modern choice is to use UTF-8 source files.

```
// Typical MSVC compiler option choice:
// /utf-8
```

This improves consistency when the codebase contains non-ASCII text in literals, comments, identifiers, or test data.

30.2 Multilingual challenges

Multilingual software is not simply “English text plus more symbols.” Real international text handling introduces challenges in storage, rendering, input, comparison, normalization, sorting, file-system interaction, and API interoperability.

30.2.1 ASCII assumptions are unsafe

One of the most common beginner mistakes is assuming that one character equals one byte.
That assumption fails for modern text. In UTF-8, many characters occupy more than one byte. In UTF-16, some characters require surrogate pairs. As a result:

- byte count is not necessarily character count
- indexing by byte may split a character encoding sequence
- substring operations can become unsafe if they ignore encoding boundaries

30.2.2 Variable-length encodings

UTF-8 is compact and widely used, but it is variable-length. This means that simple numeric indexing into a byte buffer does not automatically correspond to user-visible characters.

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

int main()
{
    std::string s = u8"Hello";
    std::cout << s.size() << '\n';
}
```

For plain ASCII-only content this may look simple, but once multilingual characters appear, the number of stored bytes may differ from the number of user-perceived characters.

30.2.3 Windows API interoperability

Windows programming has historically emphasized wide-character APIs. For this reason, multilingual Windows software often has to choose between:

- UTF-8 in modern internal code

- UTF-16 for wide Win32 APIs
- explicit conversions at boundaries

This boundary design is one of the most important architectural decisions in real applications.

30.2.4 Example: wide Win32 style

```
#include <string>

int main()
{
    std::wstring title = L"";
    (void)title;
}
```

30.2.5 Conversion challenges

A multilingual application often needs to convert between representations:

- UTF-8 for files, network protocols, or cross-platform libraries
- UTF-16 for Windows GUI or Win32 API calls
- locale-sensitive formats for user-facing display

Such conversions must be designed carefully because incorrect assumptions can corrupt text or lose data.

30.2.6 Normalization and equivalence

Unicode text may have multiple valid encoded forms for visually equivalent content. Therefore, plain binary comparison is not always sufficient for higher-level text logic.
For example, text equality in a user-facing application may require more than comparing raw stored units. This is especially important in search, indexing, and internationalized input handling.

30.2.7 Sorting and collation

Alphabetical order is language-dependent. A simple byte-wise or code-point-wise comparison may not produce correct linguistic ordering for multilingual users.
In practical software architecture, this means:

- internal storage format and display order are different concerns
- binary comparison and linguistic comparison are not the same
- UI sorting should usually be delegated to locale-aware or platform-aware facilities

30.2.8 Console output issues on Windows

Console text output on Windows deserves special attention. Even if the program stores UTF-8 correctly, console rendering depends on terminal support, code page configuration, font support, and whether narrow or wide output is used.
For this reason, a string being valid in memory does not automatically guarantee correct display in every terminal environment.

30.2.9 File names and paths

Multilingual applications must also handle file names and directory paths correctly. On Windows, Unicode-capable APIs are essential because file names may contain text from many languages.
In modern C++, `std::filesystem::path` is especially valuable because it integrates with platform path handling more safely than ad hoc string code.

```
#include <filesystem>

int main()
{
    std::filesystem::path p = L"C:\\Data\\.txt";
    (void)p;
}
```

30.2.10 User input and text boundaries

Languages differ not only in letters, but also in punctuation, directionality, shaping, combining marks, and segmentation rules. This means that robust multilingual applications must distinguish between:

- storage representation
- rendering behavior
- editing behavior
- user-perceived text boundaries

This is one reason why text handling should be treated as a first-class design topic rather than as a minor implementation detail.

30.3 Modern string handling

Modern C++ string handling should be based on clear encoding decisions, explicit ownership, safe interfaces, and clean boundaries between internal text storage and platform APIs.

30.3.1 Choose an internal text strategy

A strong modern design usually begins by choosing one main internal representation for text.
Common strategies include:

- UTF-8 internally for portability and compact storage
- UTF-16 internally in Windows-heavy applications

- explicit conversion at system boundaries

The key is consistency. A codebase becomes fragile when different modules silently assume different encodings for the same type.

30.3.2 Use the standard string classes appropriately

Modern C++ provides several standard string types:

- `std::string`
- `std::wstring`
- `std::u8string`
- `std::u16string`
- `std::u32string`

Each corresponds to a different character type.

30.3.3 UTF-8-oriented string example

```
#include <string>

int main()
{
    std::u8string name = u8"Modern C++";
    (void)name;
}
```

30.3.4 Wide string example for Windows APIs

```
#include <string>

int main()
{
    std::wstring ws = L"Windows Unicode";
    (void)ws;
}
```

30.3.5 String views for non-owning access

Modern C++ code should often separate ownership from observation. For non-owning text parameters, string views are useful.

```
#include <iostream>
#include <string_view>

void print_text(std::string_view text)
{
    std::cout << text << '\n';
}
```

```
int main()
{
    std::string_view sv = "Hello";
    print_text(sv);
}
```

This avoids unnecessary copying and communicates that the function does not take ownership.

30.3.6 UTF-8-aware API boundary design

If a codebase stores UTF-8 internally, a typical Windows-oriented design may look like this:

- keep internal business logic in UTF-8 strings
- convert to wide strings when calling Win32 wide APIs
- convert back only when necessary

This keeps encoding assumptions localized instead of scattered.

30.3.7 Avoid raw byte assumptions in string algorithms

Many naive string algorithms are unsafe for multilingual text. Examples of risky assumptions include:

- indexing by byte as if it meant one character
- truncating arbitrary byte positions
- using plain `toupper` or `tolower` logic for international text rules
- comparing strings only by byte sequence when user-facing equivalence matters

30.3.8 Use raw string literals when readability helps

Raw string literals can improve readability when text contains many backslashes or quotes.

```
#include <string>

int main()
{
    std::string path = R"(C:\Temp\notes.txt)";
    (void)path;
}
```

This is especially useful in Windows path examples, regular expressions, JSON fragments, and test data.

30.3.9 Universal character names

C++ also supports universal character names, which can express Unicode code points portably inside source text.

```
#include <iostream>

int main()
{
    const char* text = "A:\u0041";
    std::cout << text << '\n';
}
```

This can be useful when a source environment cannot directly contain a particular character or when explicit code-point notation is desired.

30.3.10 A practical UTF-8 configuration example

For a Windows-focused modern C++ project, a good practical setup often includes:

- UTF-8 encoded source files
- MSVC `/utf-8`
- explicit use of `u8` literals where UTF-8 intent matters
- well-defined conversion boundaries to Win32 wide APIs

30.3.11 Example: modern text parameter design

```
#include <string>
#include <string_view>

class Document
{
private:
    std::string title_;

public:
    explicit Document(std::string title)
        : title_(std::move(title))
    {
    }

    std::string_view title() const noexcept
    {
        return title_;
    }
};

int main()
{
    Document doc("Modern C++");
```

```
    auto view = doc.title();
    (void)view;
}
```

This example shows a strong general design principle:

- own text with a standard string type
- expose read-only views when ownership transfer is not needed

30.3.12 Prefer explicit encoding over ambiguous conventions

A professional codebase should avoid ambiguous questions such as:

> Is this `std::string` ASCII, local code page text, UTF-8, or raw bytes?

Instead, documentation, naming, and interfaces should make the expectation clear. That clarity becomes especially important in multilingual systems.

30.4 Windows-oriented practical examples

30.4.1 Example 1: UTF-8 text in modern C++20 code

```
#include <string>

int main()
{
    std::u8string greeting = u8"Hello";
    std::u8string arabic   = u8"";
    std::u8string japanese = u8"";

    (void)greeting;
    (void)arabic;
    (void)japanese;
}
```

30.4.2 Example 2: wide string for Windows-facing code

```
#include <string>

int main()
{
    std::wstring file_name = L".txt";
    (void)file_name;
}
```

30.4.3 Example 3: safe non-owning string access

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

```cpp
#include <string_view>

void log_message(std::string_view message)
{
    std::cout << message << '\n';
}

int main()
{
    std::string text = "Build completed";
    log_message(text);
}
```

30.4.4 Example 4: raw string literal for path-like text

```cpp
#include <string>

int main()
{
    std::string config = R"({
    "root": "C:\Projects\Data"
})";
    (void)config;
}
```

30.5 Best practices

30.5.1 Use UTF-8 consciously

UTF-8 is a strong default for modern cross-platform software because it is compact, widely used, and interoperates well with many file formats, protocols, and tools.

30.5.2 Keep platform conversions at boundaries

If Windows APIs require wide strings, keep conversion logic near those API boundaries rather than spreading it throughout the program.

30.5.3 Prefer standard string ownership types

Use:

- `std::string`
- `std::wstring`
- `std::u8string`
- `std::string_view`

according to the intended ownership and encoding design.

30.5.4 Do not assume one byte equals one character

This assumption breaks quickly in multilingual software.

30.5.5 Make encoding expectations explicit

A team should be able to answer clearly what encoding a given string type represents in each subsystem.

30.5.6 Use Unicode-capable APIs on Windows

For multilingual Windows software, Unicode-capable APIs are the practical modern choice.

30.6 Common beginner mistakes

30.6.1 Mistake 1: treating text as raw bytes without an encoding model

This makes bugs inevitable once multilingual input appears.

30.6.2 Mistake 2: assuming `std::string` means UTF-8 automatically

It may contain UTF-8, but the type itself does not guarantee that unless the codebase defines and follows that convention.

30.6.3 Mistake 3: mixing encodings silently

Combining UTF-8, UTF-16, and locale-dependent narrow strings without explicit boundaries leads to corruption and debugging difficulty.

30.6.4 Mistake 4: indexing multilingual text as if each element were a full character

That is unsafe for variable-length encodings.

30.6.5 Mistake 5: ignoring Windows API expectations

On Windows, text interoperability often requires wide-string awareness.

Summary

Modern text handling in C++ requires understanding both language facilities and encoding realities. The core ideas are:

- text is not just bytes
- encoding and storage type must be chosen deliberately
- multilingual software requires explicit design, not assumptions
- modern C++ offers distinct character types and string forms for different text models
- Windows-oriented code often benefits from a clear UTF-8 internal strategy plus explicit UTF-16 API boundaries

A professional C++ programmer should treat Unicode and text encoding as architectural concerns. Once the encoding model is made explicit and the string handling strategy is consistent, modern C++ becomes far safer and clearer for multilingual applications than older ad hoc string code.

Part IX

Writing Modern and Safe C++

Best Practices

31.1 Use auto wisely

The `auto` keyword is one of the most useful features in modern C++, but like many powerful tools, it should be used with judgment. Its main purpose is type deduction. Instead of forcing the programmer to repeat a complex or obvious type, `auto` lets the compiler deduce the type from the initializer.
Used well, `auto` improves readability, reduces repetition, avoids certain type mismatches, and makes code easier to maintain when types evolve. Used poorly, it can hide important type information and make code less clear.

31.1.1 When auto improves code

A common good use of `auto` is when the type is long, obvious from the initializer, or difficult to spell correctly.

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

int main()
{
    auto x = 42;
    auto name = std::string("Modern C++");
    auto values = std::vector<int>{1, 2, 3, 4};

    (void)x;
    (void)name;
    (void)values;
}
```

Here, `auto` removes repetition without reducing clarity.

31.1.2 Using auto with iterators

One of the strongest uses of `auto` is with iterators, because iterator types are often verbose and implementation-dependent.

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

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

```
    for (auto it = data.begin(); it != data.end(); ++it)
    {
        std::cout << *it << '\n';
    }
}
```

Without `auto`, the type may be unnecessarily long and distracting.

31.1.3 Use `auto` to avoid accidental type mismatches

A practical benefit of `auto` is that it can preserve the exact type of an expression more reliably than manually writing a guessed type.

```
#include <vector>

int main()
{
    std::vector<int> values{1, 2, 3};

    auto size = values.size();

    (void)size;
}
```

This avoids mistakes such as forcing `int` when the actual type returned is a container size type.

31.1.4 Use `auto` with complex expressions

```
#include <map>
#include <string>

int main()
{
    std::map<int, std::string> records;
    auto result = records.insert({1, "Compiler"});

    (void)result;
}
```

The deduced type here is lengthy and not worth repeating manually.

31.1.5 When `auto` should be used carefully

`auto` should not be used mechanically everywhere. A reader should still be able to understand what the variable represents. Poor use:

```
auto value = some_function();
```

If the initializer gives no clue about the result type and the exact type matters to understanding, then a more explicit spelling may be better.

31.1.6 A good practical rule

Use `auto` when:

- the type is obvious from the initializer
- the type is long or noisy
- exact compiler deduction helps avoid mismatch
- the name communicates the role clearly

Be more explicit when:

- the precise type is important to understanding
- deduction may hide narrowing or ownership intent
- the initializer does not reveal enough meaning

31.1.7 Preserving references and cv-qualification

A very important detail is that plain `auto` does not always preserve references or top-level constness in the way beginners expect.

```
#include <iostream>

int main()
{
    int x = 10;
    int& ref = x;

    auto a = ref;   // a is int
    auto& b = ref;  // b is int&

    b = 20;
    std::cout << x << '\n';
}
```

This is why `auto`, `auto\&`, and `const auto\&` should be chosen deliberately.

31.1.8 Range-based for loops

Modern C++ often combines `auto` with range-based loops.

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

int main()
{
    std::vector<int> numbers{1, 2, 3, 4};

    for (const auto& n : numbers)
    {
```

```
        std::cout << n << '\n';
    }
}
```

This is a strong modern style:

- `const` because elements are only read
- reference to avoid unnecessary copying
- `auto` to avoid noisy type spelling

31.2 Prefer STL containers

The Standard Template Library containers are among the safest and most important foundations of modern C++. In most application code, standard containers should be preferred over manual arrays, home-grown memory-managed structures, or raw ownership patterns.
Why they should be preferred:

- they manage memory automatically
- they express intent clearly
- they integrate well with algorithms
- they reduce bugs related to bounds, lifetime, and cleanup
- they provide tested and standardized behavior

31.2.1 Prefer `std::vector` over raw dynamic arrays

Old style:

```
int* data = new int[100];
// ...
delete[] data;
```

Modern style:

```
#include <vector>

int main()
{
    std::vector<int> data(100);
}
```

The `std::vector` version is safer because:

- it releases memory automatically
- it knows its size
- it supports iteration and algorithms directly

31.2.2 Use the container that matches the intent

Some common choices are:

- `std::vector` for dynamic contiguous sequences
- `std::array` for fixed-size arrays
- `std::string` for text
- `std::map` or `std::unordered_map` for key-value storage
- `std::set` for unique ordered elements
- `std::deque` when front and back growth both matter

31.2.3 Example: fixed-size data with `std::array`

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

int main()
{
    std::array<int, 4> values{10, 20, 30, 40};

    for (int v : values)
    {
        std::cout << v << '\n';
    }
}
```

31.2.4 Example: associative storage with `std::map`

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

int main()
{
    std::map<int, std::string> users;
    users[1] = "Ayman";
    users[2] = "Developer";

    std::cout << users[1] << '\n';
}
```

31.2.5 Containers plus algorithms are a modern style

A major strength of STL containers is their integration with algorithms.

```
#include <algorithm>
#include <vector>
```

```cpp
#include <iostream>

int main()
{
    std::vector<int> values{5, 2, 9, 1, 3};

    std::sort(values.begin(), values.end());

    for (const auto& v : values)
    {
        std::cout << v << ' ';
    }
    std::cout << '\n';
}
```

This is usually clearer and safer than hand-written low-level loops and manual memory logic.

31.3 Avoid raw memory management

One of the strongest best practices in modern C++ is to avoid owning raw pointers and direct `new`/`delete` whenever possible. Manual memory management is error-prone because it easily leads to:

- memory leaks
- double deletion
- dangling pointers
- exception-unsafety
- ownership confusion

31.3.1 Prefer RAII

Resources in modern C++ should usually be owned by objects whose destructors handle cleanup automatically.
Examples include:

- `std::vector`
- `std::string`
- `std::unique_ptr`
- `std::shared_ptr`
- file stream classes
- lock guards

31.3.2 Avoid raw owning pointers

Unsafe style:

```
class BadOwner
{
private:
    int* data_;

public:
    BadOwner()
        : data_(new int[100])
    {
    }

    ~BadOwner()
    {
        delete[] data_;
    }
};
```

This style is fragile because copy and move operations must now be designed carefully.

31.3.3 Prefer standard ownership tools

Better style:

```
#include <memory>

class GoodOwner
{
private:
    std::unique_ptr<int[]> data_;

public:
    GoodOwner()
        : data_(std::make_unique<int[]>(100))
    {
    }
};
```

Or even better, if the intent is simply “a growable sequence of integers,” use a container:

```
#include <vector>

class BetterOwner
{
private:
    std::vector<int> data_;

public:
    BetterOwner()
        : data_(100)
```

```
    {
    }
};
```

31.3.4 Use `std::make_unique` and `std::make_shared`

When dynamic allocation is truly needed, prefer factory helpers.

```
#include <memory>

int main()
{
    auto p1 = std::make_unique<int>(42);
    auto p2 = std::make_shared<int>(100);

    (void)p1;
    (void)p2;
}
```

This improves clarity and exception safety.

31.3.5 Raw pointers are often fine as non-owning observers

Modern guidance does not mean raw pointers are forbidden. A raw pointer is often acceptable when it clearly represents a non-owning observer.

```
#include <iostream>

void print_value(const int* p)
{
    if (p)
    {
        std::cout << *p << '\n';
    }
}
```

The key idea is that ownership should not be represented by raw pointers in modern code.

31.4 Use `nullptr`

Modern C++ provides `nullptr` as the proper null pointer literal. It should be preferred over `0` and `NULL`.

31.4.1 Why `nullptr` is better

Before C++11, null pointers were often written as:

```
int* p = 0;
```

or

```
int* p = NULL;
```

These are weaker choices because they can behave as integers in overload resolution and template deduction.
`nullptr` has its own dedicated type and represents null pointer intent clearly.

31.4.2 Basic use

```
int* p = nullptr;
```

31.4.3 Safer overload resolution

```
#include <iostream>

void f(int)
{
    std::cout << "int overload\n";
}

void f(int*)
{
    std::cout << "pointer overload\n";
}

int main()
{
    f(nullptr);
}
```

Here, `nullptr` selects the pointer overload cleanly.

31.4.4 Clearer intent

`nullptr` tells the reader immediately:

> This value is meant to be a null pointer, not an integer zero.

31.4.5 Use `nullptr` in modern pointer checks

```
#include <iostream>

int main()
{
    int* p = nullptr;

    if (p == nullptr)
    {
        std::cout << "No object\n";
    }
}
```

In many cases, direct boolean checks are also acceptable:

```
if (!p)
{
}
```

31.5 Use `override`

When a derived class is intended to override a virtual function from a base class, use the `override` specifier.
This is one of the simplest and most valuable correctness tools in modern C++.

31.5.1 Why `override` matters

Without `override`, a programmer may accidentally create a new function instead of overriding the intended base virtual function.
This can happen because of:

- wrong parameter type
- missing const qualifier
- wrong reference qualifier
- spelling mistakes

31.5.2 Unsafe style

```
struct Base
{
    virtual void draw() const
    {
    }
};

struct Derived : Base
{
    void draw()
    {
    }
};
```

This does not override the base function because the const qualifier differs.

31.5.3 Safe modern style

```
struct Base
{
    virtual void draw() const
    {
    }
};

struct Derived : Base
{
    void draw() const override
    {
    }
};
```

Now the compiler verifies that a real override happens.

31.5.4 Use `override` consistently

Whenever a derived virtual function is meant to override, mark it.

```
#include <iostream>

class Animal
{
public:
    virtual ~Animal() = default;

    virtual void speak() const
    {
        std::cout << "Animal sound\n";
    }
};

class Dog : public Animal
{
public:
    void speak() const override
    {
        std::cout << "Woof\n";
    }
};
```

31.5.5 Benefits

Using `override` gives:

- stronger correctness checking
- clearer design intent
- better maintainability during refactoring

31.6 Prefer `const`

Const-correctness is a foundational best practice in modern C++. It communicates intent, prevents accidental modification, and improves program safety.
A `const` object or reference expresses:

> This value should not be modified through this name.

31.6.1 Use `const` for values that should not change

```
#include <iostream>
```

```
int main()
{
    const int max_connections = 100;
    std::cout << max_connections << '\n';
}
```

This prevents accidental reassignment.

31.6.2 Use `const` references for read-only parameters

Passing expensive objects by `const` reference avoids copying and communicates that the function will not modify the argument.

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

void print_name(const std::string& name)
{
    std::cout << name << '\n';
}
```

31.6.3 Use `const` member functions

If a member function does not modify the logical state of the object, mark it `const`.

```
#include <string>

class User
{
private:
    std::string name_;

public:
    explicit User(std::string name)
        : name_(std::move(name))
    {
    }

    const std::string& name() const
    {
        return name_;
    }
};
```

This allows the function to be called on const objects and improves interface clarity.

31.6.4 Use `const auto&` when reading elements

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

int main()
{
```

```
    std::vector<int> values{1, 2, 3};

    for (const auto& value : values)
    {
        std::cout << value << '\n';
    }
}
```

This avoids copying while preserving read-only intent.

31.6.5 Const improves reasoning

Const-correct code is easier to reason about because fewer names are allowed to mutate state. This becomes especially important in large codebases and APIs.

31.6.6 Do not confuse low-level and top-level const

Modern C++ programmers should understand that const can apply to:

- the object itself
- the pointee
- both

```
int value = 10;
const int* p1 = &value; // cannot modify through p1
int* const p2 = &value; // p2 cannot point elsewhere
const int* const p3 = &value; // both restrictions
```

31.7 Avoid macros

Macros belong to the preprocessor, not the C++ type system. They have legitimate specialized uses such as header guards, conditional compilation, and certain low-level build-time integrations, but they should generally be avoided for ordinary programming tasks.

31.7.1 Why macros are risky

Macros are problematic because they:

- are not type-safe
- ignore scope rules
- can produce surprising substitutions
- are hard to debug
- can hide logic and errors

31.7.2 Classic bad macro example

```
#define SQUARE(x) x * x
```

This is dangerous:

```
int value = SQUARE(1 + 2);
```

After expansion, the result becomes:

```
int value = 1 + 2 * 1 + 2;
```

which is not the intended square operation.

31.7.3 Prefer constexpr functions

Modern C++ provides better alternatives.

```
constexpr int square(int x)
{
    return x * x;
}

int main()
{
    int value = square(1 + 2);
    (void)value;
}
```

This version is type-safe, scoped, debuggable, and semantically correct.

31.7.4 Prefer constants over object-like macros

Bad style:

```
#define BUFFER_SIZE 1024
```

Modern style:

```
constexpr int buffer_size = 1024;
```

31.7.5 Prefer templates over function-like macros

Bad style:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Better style:

```
template <typename T>
constexpr const T& max_value(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

31.7.6 Acceptable macro uses

Macros still have limited practical roles such as:

- include guards
- conditional compilation
- compiler/platform feature toggles
- certain low-level interoperability requirements

Even then, use them carefully and keep them localized.

31.8 Practical combined example

The following example demonstrates several modern best practices together.

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

class Shape
{
public:
    virtual ~Shape() = default;

    virtual void draw() const
    {
        std::cout << "Shape\n";
    }
};

class Circle : public Shape
{
public:
    void draw() const override
    {
        std::cout << "Circle\n";
    }
};

int main()
{
    const std::string title = "Modern C++";
    std::vector<std::string> names{"Ayman", "C++", "Windows"};

    auto shape = std::make_unique<Circle>();

    for (const auto& name : names)
```

```
    {
        std::cout << name << '\n';
    }

    if (shape != nullptr)
    {
        shape->draw();
    }

    std::cout << title << '\n';
}
```

This example applies:

- `const` for immutable text
- STL containers instead of raw arrays
- `auto` where deduction is clear
- smart pointer ownership instead of raw memory management
- `nullptr` for null comparison
- `override` for safe virtual overriding

Summary

Writing modern and safe C++ is not about chasing fashionable syntax. It is about expressing intent clearly, reducing classes of bugs, and working with the language and standard library in their strongest form.
The practices in this chapter reinforce that goal:

- use `auto` where it improves clarity and correctness
- prefer standard containers over manual storage management
- avoid raw memory ownership and use RAII-based tools
- use `nullptr` instead of old null pointer forms
- mark intended overrides with `override`
- prefer `const` to make intent and safety explicit
- avoid macros when modern language alternatives exist

A programmer who follows these practices writes code that is easier to understand, safer to maintain, and more aligned with the design philosophy of modern C++.

What to Avoid from Old C++

32.1 Raw arrays

Raw arrays are part of the language and still have valid low-level uses, but they should not be the default data structure in modern application code. In older C and early C++ styles, arrays such as `int data[100]` or manually allocated buffers such as `new int[n]` were used very widely. Modern C++ prefers safer abstractions because raw arrays lose size information easily, decay to pointers in many expressions, and integrate poorly with modern ownership and safety rules.

32.1.1 Why raw arrays are problematic

A raw array has several weaknesses in ordinary application code:

- it does not behave like a first-class value type
- it often decays to a pointer, losing size information
- bounds are not checked by default
- copying and assignment behave differently than many beginners expect
- it is easy to separate the data from the length accidentally

Consider this example:

```
#include <iostream>

void print_values(const int* data, std::size_t count)
{
    for (std::size_t i = 0; i < count; ++i)
    {
        std::cout << data[i] << ' ';
    }
    std::cout << '\n';
}

int main()
{
    int values[4] = {10, 20, 30, 40};
    print_values(values, 4);
}
```

This works, but the function signature has already lost the fact that the original object was an array of exactly four integers. Inside the function, only a pointer plus a separate size remains. If the wrong size is passed, the function may read beyond the valid range.

32.1.2 Array decay is a major source of bugs

One of the classic problems with raw arrays is array-to-pointer decay. In many expressions, the array type is not preserved.

```
#include <iostream>

void inspect(int* p)
{
    std::cout << "Pointer received\n";
}

int main()
{
    int values[5] = {1, 2, 3, 4, 5};
    inspect(values);
}
```

Although `values` is an array, the called function only sees a pointer. This means:

- the element count is lost
- the compiler cannot help as much with size correctness
- the interface no longer documents the real shape of the data

32.1.3 Prefer `std::array` for fixed-size data

When the size is known at compile time, `std::array` is usually a much better choice.

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

void print_values(const std::array<int, 4>& data)
{
    for (int value : data)
    {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::array<int, 4> values = {10, 20, 30, 40};
    print_values(values);
}
```

Advantages:

- size is part of the type
- the container behaves like a normal object
- it integrates well with algorithms
- copying and assignment are well-defined

32.1.4 Prefer `std::vector` for dynamic-size data

When the size is not fixed at compile time, `std::vector` is usually the correct default.

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

int main()
{
    std::vector<int> values{10, 20, 30, 40};

    for (int value : values)
    {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

32.1.5 Use `std::span` for non-owning views

When a function should accept an existing sequence without taking ownership, `std::span` is often the safest modern interface.

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

void print_values(std::span<const int> values)
{
    for (int value : values)
    {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::vector<int> v{1, 2, 3, 4};
    print_values(v);

    int raw[3] = {7, 8, 9};
    print_values(raw);
}
```

This is a strong modern design because it keeps size and pointer together in one view type.

32.1.6 When raw arrays are still reasonable

Raw arrays are not forbidden. They still have limited use cases such as:

- extremely low-level code
- compatibility with C interfaces
- fixed internal storage in carefully designed low-level components
- compile-time string literals and legacy APIs

But in ordinary modern C++ application code, they should not be the first choice.

32.2 Manual memory management

Manual memory management with `new`, `delete`, `new[]`, and `delete[]` was once common, but modern C++ strongly discourages using it directly in normal code.

32.2.1 Why manual memory management is dangerous

Direct memory ownership using raw pointers creates several common problems:

- forgetting to release memory causes leaks
- deleting twice causes undefined behavior
- dereferencing after deletion causes dangling-pointer bugs
- exceptions can bypass manual cleanup paths
- ownership is often unclear at interfaces

32.2.2 A classic leak-prone pattern

```
#include <stdexcept>

void process(bool fail)
{
    int* data = new int[100];

    if (fail)
    {
        throw std::runtime_error("operation failed");
    }

    delete[] data;
}
```

If the exception is thrown, the allocated memory is never released.

32.2.3 Manual ownership makes copy semantics harder

As soon as a class owns raw memory directly, the programmer must think carefully about:

- destructor behavior
- copy constructor
- copy assignment
- move constructor
- move assignment

Example of a fragile class:

```
class BadBuffer
{
private:
    int* data_;

public:
    BadBuffer()
        : data_(new int[100])
    {
    }

    ~BadBuffer()
    {
        delete[] data_;
    }
};
```

This class looks simple, but copying it with compiler-generated defaults is dangerous because the pointer value may be copied rather than the memory ownership being duplicated correctly.

32.2.4 Prefer RAII and standard library ownership types

Modern C++ uses RAII to bind resource lifetime to object lifetime.
Prefer:

- `std::vector` for dynamic arrays
- `std::string` for text
- `std::unique_ptr` for exclusive ownership
- `std::shared_ptr` only when shared ownership is truly required

32.2.5 Modern replacement using `std::vector`

```
#include <vector>
#include <stdexcept>

void process(bool fail)
{
    std::vector<int> data(100);

    if (fail)
    {
        throw std::runtime_error("operation failed");
    }
}
```

Now cleanup is automatic even when exceptions occur.

32.2.6 Modern replacement using `std::unique_ptr`

```
#include <memory>

int main()
{
    auto value = std::make_unique<int>(42);
}
```

This expresses exclusive ownership clearly and eliminates explicit `delete`.

32.2.7 Raw pointers are still fine as observers

Modern guidance does not mean raw pointers are always wrong. A raw pointer is often acceptable when it is clearly non-owning.

```
#include <iostream>

void print_value(const int* p)
{
    if (p)
    {
        std::cout << *p << '\n';
    }
}
```

The key distinction is:

- raw pointers are often acceptable for observation
- raw pointers are poor tools for ownership

32.3 C-style casts

C-style casts are one of the most discouraged features inherited from older C and early C++ style. They are too broad, too implicit in meaning, and too easy to misuse.

32.3.1 Why C-style casts are problematic

A cast written like this:

```
int x = (int)value;
```

or

```
int x = int(value);
```

does not clearly communicate what kind of conversion is being requested.
In C++, a C-style cast may act like one of several very different mechanisms, including conversions similar to:

- `static_cast`
- `const_cast`
- `reinterpret_cast`

This makes it harder for readers and tools to reason about the code.

32.3.2 Prefer named C++ casts

Modern C++ provides named cast operators that make intent explicit.

- `static_cast` for ordinary checked conversions
- `dynamic_cast` for polymorphic downcasts and safe runtime-checked conversions
- `const_cast` for explicit cv-qualification changes
- `reinterpret_cast` for low-level bit reinterpretation, used only with extreme caution

32.3.3 Example: use `static_cast` for numeric conversion

Old style:

```
double value = 3.9;
int x = (int)value;
```

Modern style:

```
double value = 3.9;
int x = static_cast<int>(value);
```

This is clearer because it signals a normal explicit conversion.

32.3.4 Example: use `dynamic_cast` for polymorphic downcasting

Unsafe old-style idea:

```
struct Base { virtual ~Base() = default; };
struct Derived : Base { void run() {} };

void f(Base* p)
{
    Derived* d = (Derived*)p;
    d->run();
}
```

Safer modern style:

```
struct Base { virtual ~Base() = default; };
struct Derived : Base { void run() {} };

void f(Base* p)
{
    if (auto d = dynamic_cast<Derived*>(p))
    {
        d->run();
    }
}
```

This makes the downcast explicit and checked.

32.3.5 Example: avoid hidden dangerous reinterpretation

Old-style casts can accidentally hide a reinterpret-style conversion that is risky and hard to review.
When low-level reinterpretation is truly needed, it should be written explicitly so the risk is visible:

```
#include <cstdint>

void* raw = nullptr;
auto p = reinterpret_cast<std::uintptr_t>(raw);
(void)p;
```

Even then, such code should be rare and justified.

32.3.6 A practical rule

If a cast is needed:

- use the named cast that matches the intent
- avoid the cast entirely if redesign can remove it
- treat `reinterpret_cast` and `const_cast` as last-resort tools

32.4 NULL

The old macro `NULL` belongs to older C and C++ style. In modern C++, it should be replaced by `nullptr`.

32.4.1 Why `NULL` is outdated

Historically, `NULL` was often defined as an integer literal such as `0`. That created ambiguity because integer zero is not the same thing as a dedicated null pointer value in the type system.
Modern C++ introduced `nullptr`, which has the type `std::nullptr_t` and is specifically meant to represent a null pointer constant.

32.4.2 Old style

```
int* p = NULL;
```

32.4.3 Modern style

```
int* p = nullptr;
```

32.4.4 Why `nullptr` is better

Advantages of `nullptr`:

- it clearly expresses null pointer intent
- it participates correctly in overload resolution
- it avoids confusion with integer zero
- it is the modern standard form

32.4.5 Overload example

```
#include <iostream>

void f(int)
{
    std::cout << "int overload\n";
}

void f(int*)
{
    std::cout << "pointer overload\n";
}

int main()
{
    f(nullptr);
}
```

This clearly calls the pointer overload.

32.4.6 Use `nullptr` throughout modern code

Examples:

```
int* p = nullptr;

if (p == nullptr)
{
}
```

and often simply:

```
if (!p)
{
}
```

In modern C++, new code should not introduce `NULL`.

32.5 Overuse of inheritance

Inheritance is one of the classic object-oriented tools in C++, but older C++ code often overused it. Modern C++ is much more cautious. Inheritance is powerful when modeling a true base-to-derived relationship with substitutability, but it is often the wrong default tool for code reuse.

32.5.1 Why inheritance was overused

In older design styles, inheritance was sometimes used for:

- code reuse only
- access to base implementation details
- organizing classes in large taxonomies
- replacing composition with deep hierarchies

This often produced designs that were fragile, tightly coupled, and hard to change.

32.5.2 Problems caused by too much inheritance

Overuse of inheritance can lead to:

- deep and confusing class hierarchies
- tight coupling between base and derived classes
- brittle virtual interfaces
- hard-to-predict effects of base-class changes
- misuse of inheritance where composition would be clearer

32.5.3 Inheritance should model an “is-a” relationship

Good inheritance generally requires a real substitutability relationship.
For example, if a derived class can be used anywhere a base class is expected without violating meaning, inheritance may be appropriate.

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

class Shape
{
public:
    virtual ~Shape() = default;

    virtual void draw() const = 0;
};

class Circle : public Shape
{
public:
    void draw() const override
    {
        std::cout << "Drawing Circle\n";
    }
};
```

This is a good use of runtime polymorphism because the abstraction is behavioral.

32.5.4 Bad inheritance for implementation reuse

A weak design uses inheritance just to gain member functions or data layout.

```
class Logger
{
public:
    void log_message(const char* text)
    {
        (void)text;
    }
};

class Engine : public Logger
{
};
```

This says that an `Engine` *is a* `Logger`, which is conceptually questionable. Usually the relationship is instead:

> An engine *has a* logger.

32.5.5 Prefer composition when ownership or collaboration is the real relationship

Modern style:

```
class Logger
{
public:
    void log_message(const char* text)
    {
        (void)text;
    }
};

class Engine
{
private:
    Logger logger_;

public:
    void start()
    {
        logger_.log_message("Engine started");
    }
};
```

This is often simpler, clearer, and more maintainable.

32.5.6 Deep hierarchies are usually a warning sign

Large inheritance trees are often harder to reason about than flat composition-based designs. Problems increase when the hierarchy contains:

- protected state
- partially overridden behavior
- virtual calls during construction or destruction
- multiple inheritance used for ordinary code reuse

32.5.7 Use interfaces and composition deliberately

Modern C++ often prefers:

- small abstract interfaces for true polymorphism
- composition for assembling behavior
- templates and generic programming for static polymorphism
- lambdas and function objects for localized behavior

32.5.8 A practical comparison

Inheritance-heavy style:

```
class DataSource
{
public:
    virtual ~DataSource() = default;
    virtual int read() = 0;
};

class BufferedFileDataSource : public DataSource
{
public:
    int read() override
    {
        return 0;
    }
};
```

This may be fine if runtime substitutability is really needed.
But if the goal is only to combine reading logic with buffering logic internally, composition is often the better design.

```
class Buffer
{
public:
    int next()
    {
        return 0;
    }
};

class FileReader
{
private:
    Buffer buffer_;

public:
    int read()
    {
        return buffer_.next();
    }
};
```

32.6 Practical modernization examples

32.6.1 Replace raw arrays with modern containers

Old style:

```
int values[5] = {1, 2, 3, 4, 5};
```

Modern alternatives:

```
#include <array>
#include <vector>

std::array<int, 5> fixed_values{1, 2, 3, 4, 5};
std::vector<int> dynamic_values{1, 2, 3, 4, 5};
```

32.6.2 Replace owning raw pointers with RAII types

Old style:

```
int* value = new int(42);
delete value;
```

Modern style:

```
#include <memory>

auto value = std::make_unique<int>(42);
```

32.6.3 Replace C-style casts with explicit C++ casts

Old style:

```
double d = 3.5;
int x = (int)d;
```

Modern style:

```
double d = 3.5;
int x = static_cast<int>(d);
```

32.6.4 Replace `NULL` with `nullptr`

Old style:

```
int* p = NULL;
```

Modern style:

```
int* p = nullptr;
```

32.6.5 Replace inheritance-only reuse with composition

Old style:

```
class Logger
{
public:
    void log() {}
};

class App : public Logger
{
};
```

Modern style:

```
class Logger
{
public:
    void log() {}
};

class App
{
private:
    Logger logger_;
};
```

Summary

Many features of old C++ are still legal, but modern C++ encourages better defaults because experience has shown where bugs and design problems repeatedly appear.
What modern code should generally avoid from older style:

- raw arrays as the default sequence abstraction
- manual memory management with direct owning `new`/`delete`
- C-style casts that hide intent and risk
- `NULL` instead of `nullptr`
- inheritance used mainly for code reuse instead of true abstraction

What modern code should prefer instead:

- `std::array`, `std::vector`, and `std::span`
- RAII, smart pointers, and standard containers
- named C++ casts such as `static_cast` and `dynamic_cast`
- `nullptr`
- composition, small interfaces, and generic programming where appropriate

A central lesson of modern C++ is that safer code often becomes clearer code as well. The goal is not to reject the language's past, but to avoid patterns that decades of experience have shown to be fragile, error-prone, or unnecessarily hard to maintain.

Debugging and Diagnostics

33.1 Reading compiler errors

One of the most important professional skills in C++ is learning how to read diagnostics calmly and systematically. Many beginners see a long error list and assume that every line describes a different problem. In reality, a single root mistake often triggers many secondary messages.
A good debugging workflow begins with the understanding that compiler diagnostics are not random. They are the compiler's structured explanation of what failed while translating the source program into object code. The quality of diagnostics varies by compiler, but modern toolchains usually provide enough information to locate the real cause if the programmer reads them in the correct order.

33.1.1 Start from the first real error

A common mistake is jumping to the last message in the build output. In many cases, the first meaningful compiler error is the most important one, and later errors are only consequences of that first failure.
For example, a missing semicolon, a misspelled type name, or a missing include can generate many additional errors.

```
#include <iostream>

struct Point
{
    int x
    int y;
};

int main()
{
    Point p{10, 20};
    std::cout << p.x << '\n';
}
```

A compiler may report several messages, but the real source of the problem is the missing semicolon after `int x`.

33.1.2 Read the primary message, then the location

A useful habit is:

- read the first main error text

- inspect the file and line it reports
- look slightly above that line, not only at that exact line
- identify whether the error is syntax, type, declaration, or template related

Very often, the compiler notices the problem only after the real mistake has already occurred a few lines earlier.

33.1.3 Understand the difference between primary and cascading errors

Consider this example:

```
#include <string>

int main()
{
    std::strng name = "Modern C++";
}
```

The real issue is the misspelled `std::string`. If the compiler cannot parse that declaration properly, later messages may mention unexpected tokens or undeclared identifiers. The later messages are not the root cause.

33.1.4 Template diagnostics require decomposition

Template errors often look longer because the compiler must show:

- the template that was instantiated
- the types used during instantiation
- the expression that failed
- the chain of calls that led there

For template errors, a good approach is:

- find the first line that mentions your source file
- identify the template arguments involved
- look for the first failed expression
- simplify the code mentally into ordinary non-template form

33.1.5 Example: a simple template diagnostic source

```
template <typename T>
T add(T a, T b)
{
    return a + b;
}
```

```
struct NotAddable
{
};

int main()
{
    NotAddable a, b;
    auto c = add(a, b);
    (void)c;
}
```

The meaningful question is not “Why is the compiler printing many lines?” but rather “What operation inside the template is invalid for `NotAddable`?” The failed operation is `a + b`.

33.1.6 Compiler warnings matter too

Professionals do not ignore warnings casually. Warnings often identify code that compiles but is suspicious, dangerous, misleading, or non-portable.
Examples of warning-worthy issues include:

- narrowing conversions
- signed/unsigned mismatches
- uninitialized variables
- suspicious virtual overrides
- unreachable code
- missing return statements

A strong practice is to treat new warnings seriously and keep the warning level high during development.

33.1.7 Practical reading strategy

A disciplined reading order is:

1. clean the build and rebuild
2. find the first significant error in your code
3. fix one root cause at a time
4. rebuild after each meaningful fix
5. do not attempt to fix twenty cascading errors at once

33.2 Compile vs link vs runtime errors

C++ problems occur at different stages of the program lifecycle. A strong debugging mindset requires knowing which stage produced the failure, because the tools and reasoning differ.

33.2.1 Compile-time errors

Compile-time errors happen while source code is translated into object files. These errors include:

- syntax problems
- undeclared names
- type mismatches
- invalid overload resolution
- template instantiation failures

Example:

```
int main()
{
    int x = "text";
}
```

The compiler rejects this because the initializer type is invalid for `int`.

33.2.2 Typical compile-time mindset

When facing compile errors, ask:

- Is the syntax valid
- Is every name declared and visible
- Do the types match
- Are headers included correctly
- Is template usage valid for the chosen types

33.2.3 Link-time errors

Link-time errors happen after successful compilation, when the linker combines object files and libraries into the final executable or library.
These errors often involve:

- missing definitions
- unresolved external symbols
- duplicate definitions
- architecture mismatches
- incorrect project or library settings

A famous MSVC example is `LNK2019`, unresolved external symbol.

```
#include <iostream>

void greet();

int main()
{
    greet();
}
```

If `greet()` is declared but never defined anywhere in the linked program, the compiler accepts the call, but the linker fails because it cannot find the function body.

33.2.4 Another classic link-time example

```
struct Counter
{
    static int value;
};

int main()
{
    Counter::value = 10;
}
```

This may compile, but if the static data member is declared and not defined in a source file, the linker reports an unresolved external.

33.2.5 Typical link-time mindset

When facing linker errors, ask:

- Was every declared function or object actually defined
- Is the correct source file part of the build
- Is the right library linked
- Are calling conventions and signatures consistent
- Are the project architecture and libraries compatible

33.2.6 Runtime errors

Runtime errors occur after the program has built successfully and started executing. These may include:

- null dereference
- out-of-bounds access
- use-after-free

- division by zero
- invalid logic state
- failed assertions
- bad input handling

Example:

```
int main()
{
    int* p = nullptr;
    *p = 42;
}
```

The program compiles and links, but crashes or faults at runtime.

33.2.7 Logical runtime bugs

Not all runtime problems crash. Some produce wrong results.

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

This code compiles, links, and runs, but the logic is wrong.

33.2.8 A practical summary

Think of the stages like this:

- **Compile**: “Is the code legal C++?”
- **Link**: “Does every referenced symbol have a matching built definition?”
- **Runtime**: “Does the program behave correctly and safely when executed?”

33.3 Sanitizers

Sanitizers are instrumentation-based tools that detect certain classes of runtime bugs that may otherwise remain hidden or appear only intermittently. In modern C++ practice, they are among the most valuable debugging tools for memory and undefined-behavior-related defects.
On current MSVC, the primary supported sanitizer family documented prominently is AddressSanitizer.

33.3.1 What AddressSanitizer helps detect

AddressSanitizer is designed to expose many hard-to-find memory safety bugs, including patterns such as:

- heap buffer overflow
- stack buffer overflow
- use-after-free
- use-after-scope
- global buffer overflow
- certain allocation/deallocation mismatches

33.3.2 Why sanitizers are powerful

Without instrumentation, some memory bugs may appear as:

- random crashes
- corrupted data far away from the root cause
- bugs that reproduce only in some builds
- behavior that changes with optimization level

Sanitizers detect the invalid operation closer to the actual bug site.

33.3.3 Simple out-of-bounds example

```
#include <iostream>

int main()
{
    int values[3] = {10, 20, 30};
    std::cout << values[5] << '\n';
}
```

This is undefined behavior. A sanitizer-enabled build can often catch this at runtime immediately.

33.3.4 Use-after-free example

```
#include <iostream>

int main()
{
    int* p = new int(42);
    delete p;
    std::cout << *p << '\n';
}
```

A normal build may print garbage, crash later, or appear to work. A sanitizer build is much more likely to report the actual memory misuse.

33.3.5 Allocation mismatch example

```
int main()
{
    int* p = new int[10];
    delete p;
}
```

This mismatches `new[]` with `delete`. Sanitizer tooling can help catch such mistakes.

33.3.6 Windows-oriented practical note

In MSVC-based Windows development, AddressSanitizer is enabled through compiler instrumentation settings. For special checks such as allocation/deallocation mismatch, environment configuration may also matter.

33.3.7 Use sanitizers in debug and test workflows

A strong workflow is:

- build a sanitizer-enabled configuration
- run unit tests under it
- exercise boundary cases and failure paths
- fix every sanitizer finding instead of treating it as optional

33.3.8 Sanitizers do not replace reasoning

Sanitizers are powerful, but they do not prove the program is correct. They only detect bugs that are actually exercised during execution. They complement good design, good tests, and static analysis.

33.4 Static analysis

Static analysis examines code for defect patterns without relying on the program to execute those paths first. It is one of the most important tools for catching risky constructs early, especially in large codebases.

33.4.1 How static analysis differs from compilation

The compiler primarily answers:

> Is this code valid enough to compile?

Static analysis asks a different question:

> Even if this code is valid C++, does it still contain suspicious patterns likely to cause defects?

So static analysis can warn about code that is legal, but still dangerous or poor practice.

33.4.2 Examples of issues static analysis may flag

Typical findings include:

- potential null dereference
- bounds risks
- pointer arithmetic concerns
- resource leaks
- suspicious ownership patterns
- misuse of APIs
- inconsistent annotations
- guideline violations

33.4.3 MSVC code analysis

Visual Studio provides native C and C++ code analysis tools. In MSVC toolchains, the `/analyze` option enables code analysis, and rule-based checkers can report warnings distinct from ordinary compiler diagnostics.

33.4.4 Simple suspicious code example

```
#include <cstddef>

void copy_values(int* dst, const int* src, std::size_t count)
{
    for (std::size_t i = 0; i <= count; ++i)
    {
        dst[i] = src[i];
    }
}
```

This loop likely performs one write too many because of `<= count`. A static analyzer may identify the bounds risk even before runtime tests exercise it.

33.4.5 Pointer arithmetic warnings

Modern code analysis and Core Guidelines checkers often warn about patterns such as raw pointer arithmetic, because they are strongly associated with safety bugs.

```
void process(int* p)
{
    ++p;
    *p = 42;
}
```

This may be valid in context, but the analyzer encourages safer abstractions such as `std::span`.

33.4.6 Core Guidelines checkers

Visual Studio can also apply a subset of C++ Core Guidelines rules. These checks help reinforce safer modern practices such as:

- avoiding raw pointer arithmetic
- using spans for bounds-aware sequence access
- preferring safer ownership and lifetime models

33.4.7 Static analysis works best early and continuously

Static analysis is most effective when it is not treated as a one-time cleanup task. A good workflow is:

- enable analysis on build for important projects
- review warnings regularly
- suppress only after genuine justification
- keep the warning baseline healthy over time

33.4.8 Static analysis vs sanitizers

They complement each other:

- static analysis finds suspicious patterns without running the code
- sanitizers catch concrete invalid operations during execution

Use both.

33.5 Debugging mindset

Tools matter, but mindset matters even more. Strong debugging is not guessing randomly, nor is it changing many lines at once and hoping the issue disappears. Professional debugging is a disciplined process of narrowing possibilities.

33.5.1 Assume the bug is real and explainable

A helpful principle is:

> The program is doing exactly what the code and environment together imply, even if that implication is not yet understood.

This mindset keeps the investigation grounded. Avoid mystical thinking such as:

- “The compiler is broken”
- “The debugger is lying”
- “It crashes randomly for no reason”

Compiler bugs and tool bugs exist, but they are much rarer than ordinary program defects.

33.5.2 Reduce the problem

A key debugging skill is reduction. Simplify the failing case until the bug becomes easier to see.
For example:

- isolate the failing function
- comment out unrelated code
- replace external inputs with a minimal reproducer
- shrink template or generic code to the smallest failing example

33.5.3 Change one thing at a time

When debugging, avoid making many speculative changes at once. If ten changes are made together and the problem disappears, you may not know which one actually mattered.
Better process:

- form one hypothesis
- test it
- observe the result
- keep or revert based on evidence

33.5.4 Trust observations, not assumptions

If the debugger shows that a pointer is null, or a value is different than expected, do not argue with the observation. Instead ask:

- Where did this value come from
- What earlier write or control path explains it
- What invariant was expected and where was it violated

33.5.5 Use assertions to protect assumptions

Assertions are valuable because they make assumptions explicit.

```
#include <cassert>

int divide(int a, int b)
{
    assert(b != 0);
    return a / b;
}
```

If the assumption is violated during debugging, the program stops closer to the actual fault.

33.5.6 Check invariants, not only symptoms

A symptom may appear far from the real cause. For example, a crash in a sorting routine may actually be caused by memory corruption from an earlier unrelated write.
Good debugging asks:

- What should always be true here
- When did that stop being true
- What earlier event broke the invariant

33.5.7 Prefer evidence-driven debugging

Useful evidence sources include:

- compiler diagnostics
- sanitizer reports
- static analysis warnings
- debugger watches and call stacks
- logging around state transitions
- assertions and unit tests

33.5.8 Be careful with “fixes” that only hide symptoms

Examples of weak fixes:

- inserting delays into race conditions without understanding them
- catching all exceptions and ignoring them
- adding null checks everywhere instead of fixing ownership bugs
- disabling warnings instead of resolving them

A real fix addresses the cause, not only the visible failure.

33.5.9 A practical debugging checklist

When something goes wrong:

1. classify the failure: compile, link, runtime, logic
2. reproduce it reliably if possible
3. reduce to the smallest failing case

4. inspect the earliest meaningful diagnostic
5. verify assumptions with debugger, assertions, or logs
6. use sanitizer and static analysis builds
7. fix the root cause
8. rerun tests and related scenarios

33.6 Practical examples

33.6.1 Example 1: compile-time failure

```
#include <string>

int main()
{
    std::string name = 42;
}
```

Debugging question:

- Is there a valid conversion from integer to `std::string`? No.

33.6.2 Example 2: link-time failure

```
void print_message();

int main()
{
    print_message();
}
```

If no definition exists in any linked translation unit, the linker fails.
Debugging question:

- Where is the actual function body, and is that source file or library included in the build?

33.6.3 Example 3: runtime out-of-bounds defect

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

int main()
{
    std::vector<int> values{1, 2, 3};
    std::cout << values[10] << '\n';
}
```

Debugging approach:

- reproduce under debugger
- run with sanitizer if available
- inspect the index source
- ask what invariant should constrain the index

33.6.4 Example 4: logic bug

```
int multiply(int a, int b)
{
    return a + b;
}
```

This is not a compile or runtime crash problem. It is a logic defect.
Debugging approach:

- write a unit test
- compare expected and actual behavior
- inspect the implementation against the intended contract

33.6.5 Example 5: using assertions to stop earlier

```
#include <cassert>
#include <vector>

int get_element(const std::vector<int>& values, std::size_t index)
{
    assert(index < values.size());
    return values[index];
}
```

This catches invalid assumptions during debugging much earlier than silent corruption might.

33.7 Windows-oriented practical workflow

For a modern Windows C++ project using MSVC and Visual Studio, a strong practical setup often includes:

- high warning level during normal development
- debug information enabled for debug builds
- code analysis enabled on important projects
- AddressSanitizer-enabled test builds when applicable
- assertions active in debug configurations
- regular unit and integration testing

A particularly effective workflow is to maintain multiple configurations:

- ordinary Debug build
- Release build
- Debug or Release build with AddressSanitizer
- build with static analysis enabled

Different classes of defects appear more clearly in different configurations.

Summary

Debugging and diagnostics in modern C++ require both technical tools and disciplined reasoning.
A professional programmer should distinguish clearly between:

- compile-time errors, where the code is not valid enough to compile
- link-time errors, where referenced program parts are missing or incompatible
- runtime errors, where execution fails or memory safety is violated
- logic errors, where the program runs but produces wrong results

To handle these effectively, modern C++ practice relies on several complementary tools:

- compiler diagnostics for syntax and type problems
- linker diagnostics for symbol resolution and build integration issues
- sanitizers for runtime memory bug detection
- static analysis for defect patterns discovered during build
- assertions, debugger inspection, and careful reduction of the failing case

The most important habit is not memorizing tool options. It is developing an evidence-driven mindset. Read the first real error carefully, separate root causes from cascades, test one hypothesis at a time, and prefer understanding over guesswork. That mindset is one of the defining differences between random trial-and-error debugging and professional C++ problem solving.

Part X

Concurrency and Multithreading

Introduction to Concurrency

34.1 Threads

Concurrency in modern C++ allows a program to make progress on multiple tasks during overlapping periods of execution. The most familiar tool for this is the thread. A thread is an independent path of execution inside a process. Multiple threads in the same process share the same address space and many program resources, but each thread has its own execution state such as stack, instruction flow, and register context.
Modern C++ provides standard threading support through the library header:

```
#include <thread>
```

The standard thread library gives portable support for creating and managing threads without requiring direct use of platform-specific APIs.

34.1.1 What a thread is

A process may contain one or more threads. If a program starts normally, it begins with one thread, often called the main thread. Additional threads can then be created to perform work concurrently.
Typical reasons to use threads include:

- performing background work while the main thread remains responsive
- dividing independent tasks across CPU cores
- overlapping computation with input/output activity
- handling multiple client requests in a server
- separating long-running work from user-interface work

34.1.2 Basic thread creation

The standard type `std::thread` starts a new thread of execution.

```
#include <iostream>
#include <thread>

void worker()
{
    std::cout << "Worker thread is running\n";
```

```
}

int main()
{
    std::thread t(worker);
    t.join();
}
```

In this example:

- the function `worker` executes on a separate thread
- `t.join()` waits for that thread to finish
- failing to join or detach correctly would be a serious error

34.1.3 Thread creation with a lambda

A lambda is often convenient when the work is small and local.

```
#include <iostream>
#include <thread>

int main()
{
    std::thread t([]()
    {
        std::cout << "Lambda thread\n";
    });

    t.join();
}
```

34.1.4 Passing arguments to a thread

Arguments can be passed when constructing the thread.

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

void print_message(const std::string& text, int count)
{
    for (int i = 0; i < count; ++i)
    {
        std::cout << text << '\n';
    }
}

int main()
{
    std::thread t(print_message, std::string("Hello from thread"), 3);
    t.join();
}
```

34.1.5 Joining and detaching

A thread object must eventually be either joined or detached.

- `join()` means wait until the thread finishes
- `detach()` means let the thread continue independently

```
#include <thread>
#include <chrono>

void background_task()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    std::thread t(background_task);
    t.join();
}
```

Detaching should be used carefully because it becomes harder to reason about lifetime and safe shutdown.

34.1.6 Why threads matter

Threads can improve responsiveness and throughput, but they also introduce shared-state complexity. The moment multiple threads can touch the same memory, the programmer must think about synchronization, ordering, ownership, and correctness under interleaving execution.
This is why threads are powerful but dangerous. They are not simply “faster function calls.” They introduce a fundamentally more difficult programming model.

34.2 Risks and complexity

Concurrency is one of the most difficult areas in software engineering because correct multithreaded code must remain correct under many possible execution orders. A single-threaded program follows one control flow. A multithreaded program may have countless valid interleavings.
This creates risks that do not exist in ordinary sequential code.

34.2.1 Why concurrency is hard

In single-threaded code, the programmer can usually reason in a simple step-by-step way:

1. statement A runs
2. then statement B runs
3. then statement C runs

In multithreaded code, if two or more threads are involved, operations may interleave in many orders. Even if the source code is fixed, the runtime schedule may vary from one execution to another.
That means a bug may:

- appear only occasionally
- disappear when logging is added
- change behavior between Debug and Release
- depend on processor timing or system load
- become hard to reproduce reliably

34.2.2 Sources of complexity

Concurrency complexity often comes from the interaction of these issues:

- shared mutable state
- unsynchronized reads and writes
- unclear ownership and lifetime
- ordering assumptions between threads
- misuse of locks or atomics
- blocking operations and waiting dependencies

34.2.3 Common classes of concurrency bugs

Some common bug categories are:

- data races
- race conditions
- deadlocks
- livelocks
- starvation
- missed notifications
- use-after-lifetime due to detached or late-running threads

This chapter focuses on the foundational issues: threads, data races, and race conditions.

34.2.4 The cost of shared state

The most important rule for understanding concurrency is that shared mutable state is expensive in correctness. If two threads only read shared data, the situation is usually simpler. If at least one thread writes shared data, synchronization becomes essential.
For example, code like this is already dangerous:

```
#include <thread>

int counter = 0;

void increment()
{
    ++counter;
}

int main()
{
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();
}
```

This looks harmless, but it is not safe. Two threads modify the same variable without synchronization.

34.2.5 Why debugging concurrent code is harder

Ordinary logic bugs are often reproducible with the same input. Concurrency bugs may depend on timing. A program may fail only one out of a thousand runs. Adding a print statement may accidentally hide the problem by changing timing.
Because of this, concurrency debugging requires more discipline:

- reduce shared mutable state
- prefer clear ownership
- use synchronization deliberately
- keep thread interactions simple
- write small reproducible tests

34.2.6 Do not use threads where simpler designs work

Threads are not automatically the best answer. Sometimes a simpler design is better:

- a task queue
- asynchronous I/O
- futures and async abstractions

- message passing
- single-threaded event loops

The safest concurrency is often the concurrency design that minimizes direct shared-state interaction.

34.3 Data races

A data race is one of the most important and dangerous concepts in modern C++ concurrency. In general terms, a data race occurs when two or more threads access the same memory location concurrently, at least one of the accesses is a write, and the accesses are not properly synchronized.
In the C++ memory model, a data race results in undefined behavior. That means the program is no longer guaranteed to behave in any predictable or meaningful way.

34.3.1 A simple unsafe example

```
#include <iostream>
#include <thread>

int shared_value = 0;

void writer()
{
    for (int i = 0; i < 100000; ++i)
    {
        ++shared_value;
    }
}

int main()
{
    std::thread t1(writer);
    std::thread t2(writer);

    t1.join();
    t2.join();

    std::cout << shared_value << '\n';
}
```

At first glance, some beginners expect the final value always to be `200000`. But this code contains a data race.
Why?

- both threads access `shared\_value`
- both threads write to it
- there is no mutex, atomic variable, or other synchronization

The result is undefined behavior.

34.3.2 Why ++x is not atomic by default

A statement like:

```
++shared_value;
```

looks like one operation in source code, but it usually involves several lower-level steps conceptually:

1. read the current value
2. add one
3. write the new value back

If two threads interleave these steps without synchronization, one update may overwrite another.

34.3.3 Illustration of lost update

Suppose `shared\_value` is initially 10.

- Thread A reads 10
- Thread B reads 10
- Thread A computes 11
- Thread B computes 11
- Thread A writes 11
- Thread B writes 11

Two increments occurred logically, but the value increased by only one.

34.3.4 A correct version using a mutex

A mutex protects shared data by ensuring only one thread enters the protected critical section at a time.

```
#include <iostream>
#include <mutex>
#include <thread>

int shared_value = 0;
std::mutex shared_mutex;

void writer()
{
    for (int i = 0; i < 100000; ++i)
    {
        std::lock_guard<std::mutex> lock(shared_mutex);
        ++shared_value;
    }
}
```

```
int main()
{
    std::thread t1(writer);
    std::thread t2(writer);

    t1.join();
    t2.join();

    std::cout << shared_value << '\n';
}
```

This removes the data race because access to the shared variable is synchronized.

34.3.5 A correct version using an atomic

For simple counters and similar cases, an atomic type may be appropriate.

```
#include <atomic>
#include <iostream>
#include <thread>

std::atomic<int> shared_value{0};

void writer()
{
    for (int i = 0; i < 100000; ++i)
    {
        ++shared_value;
    }
}

int main()
{
    std::thread t1(writer);
    std::thread t2(writer);

    t1.join();
    t2.join();

    std::cout << shared_value << '\n';
}
```

This is safe because the shared object is atomic and the increment operation is defined accordingly.

34.3.6 Data race vs safe shared reading

If multiple threads only read the same object, and no thread writes to it, that situation is generally much safer.

```
#include <iostream>
#include <thread>
```

```cpp
const int shared_value = 42;

void reader()
{
    std::cout << shared_value << '\n';
}

int main()
{
    std::thread t1(reader);
    std::thread t2(reader);

    t1.join();
    t2.join();
}
```

There is no write here, so the usual data-race danger does not arise in the same way.

34.3.7 Data races are undefined behavior

This point must be understood clearly. In C++, a data race is not just “sometimes the wrong answer.” It is undefined behavior. That means the compiler and hardware are not required to preserve intuitive behavior. Outcomes may include:

- wrong values
- intermittent crashes
- corrupted state
- impossible-looking logic results
- behavior changing across optimization levels

34.3.8 Avoiding data races

The main strategies are:

- avoid sharing mutable state
- use mutexes for protected shared access
- use atomic types for simple atomic state
- establish clear ownership boundaries
- prefer message passing or task handoff where practical

34.4 Race conditions

A race condition is broader than a data race. A race condition happens when program correctness depends on the relative timing or ordering of operations between threads or concurrent activities.
Not every race condition is a formal data race under the C++ memory model. Some programs may be data-race-free but still logically incorrect because the order of events is not properly controlled.

34.4.1 A timing-dependent logical example

Consider a shared flag and a worker thread. Even if synchronization is added for individual accesses, the overall logic may still depend on order.

```
#include <chrono>
#include <iostream>
#include <thread>

bool ready = false;

void worker()
{
    while (!ready)
    {
    }

    std::cout << "Work started\n";
}

int main()
{
    std::thread t(worker);

    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    ready = true;

    t.join();
}
```

This example is not safe because `ready` is shared unsafely, so it has a data race. But even after fixing the data race with proper synchronization, the programmer still has to reason about timing and waiting behavior.

34.4.2 Race condition as a broader concept

A race condition is about correctness depending on who gets there first. For example:

- one thread expects data to be ready before reading it
- another thread may or may not have produced that data yet
- the program succeeds or fails depending on schedule

So:

- every data race is a serious concurrency problem
- not every race condition is a formal data race
- race conditions include higher-level ordering bugs

34.4.3 A safe but still logically sensitive pattern

Suppose a shared variable is protected by a mutex, but the logic still assumes a certain order that is not guaranteed.

```
#include <iostream>
#include <mutex>
#include <thread>

int data = 0;
bool ready = false;
std::mutex m;

void producer()
{
    std::lock_guard<std::mutex> lock(m);
    data = 42;
    ready = true;
}

void consumer()
{
    std::lock_guard<std::mutex> lock(m);
    if (ready)
    {
        std::cout << data << '\n';
    }
}
```

This may avoid a raw data race if used carefully, but if the consumer runs before the producer, nothing happens. Whether that is correct depends on the intended design. If the system requires the consumer to wait until data is ready, then a higher-level synchronization mechanism such as a condition variable is needed.

34.4.4 Race conditions often involve check-then-act logic

One classic pattern is:

1. check some condition
2. act based on that condition

If another thread changes the underlying state between the check and the act, the logic may fail.
Conceptually unsafe pattern:

```
if (!resource_in_use)
{
    use_resource();
}
```

If multiple threads can perform the same sequence without synchronization, both may observe the resource as free and both may proceed.

34.4.5 Why race conditions are dangerous

Race conditions are dangerous because they may:

- pass tests most of the time
- fail only under load
- disappear under a debugger
- depend on core count or machine speed
- appear only in production-like timing

34.4.6 Reducing race conditions

Some important strategies are:

- minimize shared mutable state
- protect invariant-changing operations as a whole
- design around ownership transfer instead of concurrent sharing
- use proper waiting primitives, not ad hoc delays
- keep thread interactions explicit and small

34.5 Practical examples

34.5.1 Example 1: simple thread launch

```
#include <iostream>
#include <thread>

void task()
{
    std::cout << "Task running on another thread\n";
}

int main()
{
    std::thread t(task);
    t.join();
}
```

34.5.2 Example 2: unsafe shared counter

```
#include <iostream>
#include <thread>

int counter = 0;

void increment_many_times()
{
    for (int i = 0; i < 10000; ++i)
    {
        ++counter;
    }
}

int main()
{
    std::thread t1(increment_many_times);
    std::thread t2(increment_many_times);

    t1.join();
    t2.join();

    std::cout << counter << '\n';
}
```

This has a data race.

34.5.3 Example 3: mutex-protected counter

```
#include <iostream>
#include <mutex>
#include <thread>

int counter = 0;
std::mutex m;

void increment_many_times()
{
    for (int i = 0; i < 10000; ++i)
    {
        std::lock_guard<std::mutex> lock(m);
        ++counter;
    }
}

int main()
{
    std::thread t1(increment_many_times);
    std::thread t2(increment_many_times);

    t1.join();
```

```
    t2.join();

    std::cout << counter << '\n';
}
```

This removes the data race.

34.5.4 Example 4: atomic counter

```
#include <atomic>
#include <iostream>
#include <thread>

std::atomic<int> counter{0};

void increment_many_times()
{
    for (int i = 0; i < 10000; ++i)
    {
        ++counter;
    }
}

int main()
{
    std::thread t1(increment_many_times);
    std::thread t2(increment_many_times);

    t1.join();
    t2.join();

    std::cout << counter << '\n';
}
```

This is often simpler for small atomic state.

34.5.5 Example 5: shared text output

Even output streams can be affected by concurrent access patterns.

```
#include <iostream>
#include <thread>

void print_a()
{
    for (int i = 0; i < 5; ++i)
    {
        std::cout << "A\n";
    }
}

void print_b()
```

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

int main()
{
    std::thread t1(print_a);
    std::thread t2(print_b);

    t1.join();
    t2.join();
}
```

Even if this does not always crash, output ordering may interleave unpredictably. This illustrates that concurrency changes expectations about sequence.

34.6 Guidelines for beginners

34.6.1 Start simple

Before learning advanced atomics or lock-free techniques, understand these basics thoroughly:

- thread lifetime
- joining
- shared state
- mutex protection
- data race meaning
- logical ordering problems

34.6.2 Prefer no sharing when possible

The easiest shared-state bug is the one avoided entirely. A safer design often passes data into a thread, lets the thread produce a result, and returns or stores that result in a controlled way.

34.6.3 Use RAII for locks

Always prefer RAII wrappers such as `std::lock_guard` instead of manual `lock()` and `unlock()` sequences.

34.6.4 Do not use sleep as synchronization

This is a common beginner mistake:

```
std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

Sleeping does not guarantee the other thread has reached the required state. Proper synchronization must be explicit.

34.6.5 Be careful with detached threads

Detached threads can outlive objects they access, which easily causes lifetime bugs. Joining is usually easier to reason about.

34.6.6 Treat concurrency bugs seriously

A small concurrency bug is not “just a little timing issue.” If there is a data race, the program already has undefined behavior.

Summary

Concurrency allows a C++ program to perform overlapping work through multiple threads, but it also introduces one of the hardest correctness challenges in programming.
The key ideas from this chapter are:

- a thread is an independent path of execution within a process
- multiple threads can improve responsiveness and throughput
- concurrency becomes difficult as soon as mutable state is shared
- a data race occurs when shared memory is accessed concurrently, at least one access is a write, and synchronization is missing
- data races are undefined behavior in C++
- a race condition is a broader timing or ordering bug whose correctness depends on execution order

For modern safe C++ concurrency, the first mental rule should be:

> Minimize shared mutable state, synchronize deliberately, and keep thread interactions as simple as possible.

This foundation is essential before moving on to mutexes, condition variables, atomics, and more advanced multithreaded design.

Concurrency Tools

35.1 `std::thread`

The class `std::thread` is the fundamental standard-library facility for starting and managing an independent thread of execution in modern C++. It is defined in the header:

```
#include <thread>
```

A `std::thread` object can be used to represent and manage one running thread. A default-constructed `std::thread` object does not represent any running thread. A `std::thread` constructed from a callable object starts a new thread and executes that callable in the new thread.

35.1.1 Basic thread creation

```
#include <iostream>
#include <thread>

void worker()
{
    std::cout << "Worker thread started\n";
}

int main()
{
    std::thread t(worker);
    t.join();
}
```

This example shows the most basic pattern:

- define a function to run
- construct a `std::thread` with that function
- call `join()` to wait for it to finish

35.1.2 Using a lambda with `std::thread`

Lambdas are often the most convenient way to launch short tasks.

```cpp
#include <iostream>
#include <thread>

int main()
{
    std::thread t([]()
    {
        std::cout << "Lambda thread is running\n";
    });

    t.join();
}
```

35.1.3 Passing arguments

Arguments can be passed to the thread entry function through the thread constructor.

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

void print_message(const std::string& text, int times)
{
    for (int i = 0; i < times; ++i)
    {
        std::cout << text << '\n';
    }
}

int main()
{
    std::thread t(print_message, std::string("Hello from std::thread"), 3);
    t.join();
}
```

35.1.4 Joinable threads

A very important property of `std::thread` is whether the thread object is *joinable*. A joinable thread object is associated with a thread of execution.

```cpp
#include <iostream>
#include <thread>

int main()
{
    std::thread t([](){});

    if (t.joinable())
    {
        std::cout << "Thread is joinable\n";
        t.join();
    }
}
```

35.1.5 Joining and detaching

A thread object must eventually be handled correctly. Two major operations exist:

- `join()` waits for the thread to finish
- `detach()` releases the association and allows the thread to continue independently

```
#include <thread>
#include <chrono>

void background_task()
{
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
}

int main()
{
    std::thread t(background_task);
    t.join();
}
```

Detached threads are more difficult to reason about because the thread may outlive objects it touches. For most beginner and intermediate designs, `join()` is easier and safer.

35.1.6 Move-only semantics

A `std::thread` object can be moved but not copied. This reflects the fact that one specific running thread should have one managing thread object.

```
#include <thread>

void task()
{
}

int main()
{
    std::thread t1(task);
    std::thread t2 = std::move(t1);

    if (t2.joinable())
    {
        t2.join();
    }
}
```

35.1.7 Thread identifiers

Each thread has an identifier of type `std::thread::id`. The current thread ID can be queried through `std::this_thread::get_id()`.

```cpp
#include <iostream>
#include <thread>

void worker()
{
    std::cout << "Worker id: " << std::this_thread::get_id() << '\n';
}

int main()
{
    std::thread t(worker);
    std::cout << "Main id: " << std::this_thread::get_id() << '\n';
    t.join();
}
```

35.1.8 Hardware concurrency

The library also provides a way to ask for an estimate of available hardware thread contexts.

```cpp
#include <iostream>
#include <thread>

int main()
{
    std::cout << "hardware_concurrency = "
              << std::thread::hardware_concurrency() << '\n';
}
```

This value is only an estimate, but it is often useful when designing worker pools or partitioning tasks.

35.2 `std::jthread`

`std::jthread` is a modern thread-management type introduced with C++20. It is closely related to `std::thread`, but it improves lifetime management and cancellation-aware design.
A practical way to think about `std::jthread` is:

- it is a thread-owning type like `std::thread`
- it is designed to work naturally with cooperative cancellation through `std::stop_token`
- it helps make thread cleanup more robust

35.2.1 Why `std::jthread` exists

One of the most common beginner mistakes with `std::thread` is forgetting to join or otherwise manage the thread correctly. `std::jthread` was introduced to encourage safer thread lifetime management.
For many application designs, `std::jthread` is the better default choice when C++20 is available.

35.2.2 Basic usage

```
#include <chrono>
#include <iostream>
#include <thread>

int main()
{
    std::jthread t([]()
    {
        std::cout << "jthread started\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        std::cout << "jthread finished\n";
    });
}
```

This is intentionally simple. The main difference for beginners is that `std::jthread` is designed to reduce lifetime-management mistakes.

35.2.3 Using `std::stop_token`

A major feature of `std::jthread` is support for cooperative stop requests. A thread function can accept a `std::stop_token` and check whether a stop was requested.

```
#include <chrono>
#include <iostream>
#include <stop_token>
#include <thread>

void looping_task(std::stop_token st)
{
    while (!st.stop_requested())
    {
        std::cout << "Working...\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    std::cout << "Stop requested\n";
}

int main()
{
    std::jthread t(looping_task);

    std::this_thread::sleep_for(std::chrono::milliseconds(300));
    t.request_stop();
}
```

This style is much cleaner than many older ad hoc flag-based approaches.

35.2.4 Why cooperative stop matters

A stop request does not forcibly kill the thread. Instead, it provides a structured way for the thread to observe a request and end its work cleanly. This is much safer than trying to terminate threads abruptly.

35.2.5 When to prefer `std::jthread`

Prefer `std::jthread` when:

- C++20 is available
- the thread has clear ownership and scope
- cooperative stop handling is useful
- you want safer automatic thread-lifetime management

35.3 `std::mutex`

A mutex is the standard tool for protecting shared mutable state. It provides mutual exclusion, meaning that only one thread at a time can own the lock and execute the protected critical section.
The primary standard header is:

```
#include <mutex>
```

35.3.1 What a mutex does

A mutex protects code that accesses shared state. Without synchronization, multiple threads writing or reading-and-writing the same object can create data races and undefined behavior.
A mutex provides these core operations:

- `lock()`
- `try_lock()`
- `unlock()`

35.3.2 Unsafe shared counter

```
#include <iostream>
#include <thread>

int counter = 0;

void increment()
{
    for (int i = 0; i < 10000; ++i)
    {
        ++counter;
    }
```

```
}

int main()
{
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << counter << '\n';
}
```

This code has a data race.

35.3.3 Safe version with `std::mutex`

```
#include <iostream>
#include <mutex>
#include <thread>

int counter = 0;
std::mutex counter_mutex;

void increment()
{
    for (int i = 0; i < 10000; ++i)
    {
        std::lock_guard<std::mutex> lock(counter_mutex);
        ++counter;
    }
}

int main()
{
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << counter << '\n';
}
```

The critical section is now protected by the mutex.

35.3.4 Why RAII locking is essential

Although a mutex exposes `lock()` and `unlock()`, modern C++ code should rarely call them manually in ordinary code. Instead, use RAII lock wrappers such as:

- `std::lock_guard`

- `std::unique_lock`
- `std::scoped_lock`

This prevents bugs caused by early returns or exceptions skipping `unlock()`.

35.3.5 Using `try_lock`

`try_lock()` attempts to acquire the mutex without blocking.

```
#include <iostream>
#include <mutex>

int main()
{
    std::mutex m;

    if (m.try_lock())
    {
        std::cout << "Mutex acquired\n";
        m.unlock();
    }
}
```

This can be useful in specialized designs, but it should be used carefully because failed lock attempts add more control-flow complexity.

35.4 `std::scoped_lock`

`std::scoped_lock` is an RAII lock wrapper introduced to simplify safe locking, especially when multiple mutexes need to be locked together.
Its key ideas are:

- lock on construction
- unlock on destruction
- support one or more mutexes

35.4.1 Basic single-mutex use

```
#include <iostream>
#include <mutex>
#include <thread>

int value = 0;
std::mutex m;

void work()
{
```

```
    std::scoped_lock lock(m);
    ++value;
}

int main()
{
    std::thread t1(work);
    std::thread t2(work);

    t1.join();
    t2.join();

    std::cout << value << '\n';
}
```

35.4.2 Why `std::scoped_lock` is useful

For a single mutex, `std::lock_guard` and `std::scoped_lock` are both useful. But `std::scoped_lock` becomes especially valuable when multiple mutexes must be acquired.

35.4.3 Locking multiple mutexes safely

```
#include <mutex>

std::mutex m1;
std::mutex m2;

void update_both()
{
    std::scoped_lock lock(m1, m2);
}
```

This is much cleaner than manual lock ordering logic and helps reduce deadlock risk when multiple mutexes are involved.

35.4.4 RAII benefit

As with other RAII lock types, the lock is released automatically when the object goes out of scope.

```
#include <iostream>
#include <mutex>

std::mutex m;

void process()
{
    std::scoped_lock lock(m);
    std::cout << "critical section\n";
}
```

This is the style that should usually be preferred in modern code.

35.5 `std::condition_variable`

A condition variable allows one or more threads to wait until some condition becomes true. It is the standard mechanism for blocking until a state change occurs, instead of repeatedly polling or sleeping.
The main header is:

```
#include <condition_variable>
```

35.5.1 When condition variables are needed

A common pattern in concurrent code is:

- one thread produces or changes state
- another thread waits until that state becomes ready

Without a condition variable, a beginner might write code that repeatedly checks a flag in a loop. That wastes CPU and is often incorrect.

35.5.2 Core operations

A condition variable provides the main operations:

- `wait()`
- `wait_for()`
- `notify_one()`
- `notify_all()`

35.5.3 Basic producer-consumer style example

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex m;
std::condition_variable cv;
bool ready = false;

void producer()
{
    {
        std::lock_guard<std::mutex> lock(m);
        ready = true;
    }
    cv.notify_one();
}
```

```
void consumer()
{
    std::unique_lock<std::mutex> lock(m);
    cv.wait(lock, []{ return ready; });

    std::cout << "Consumer sees ready = true\n";
}

int main()
{
    std::thread t1(consumer);
    std::thread t2(producer);

    t1.join();
    t2.join();
}
```

35.5.4 Why `std::unique_lock` is used

`std::condition_variable` works with `std::unique_lock<std::mutex>`. This is important because waiting temporarily unlocks and later reacquires the mutex around the blocking operation.

35.5.5 Always wait with a predicate

A best practice is to use the predicate form of `wait()`.

```
cv.wait(lock, []{ return ready; });
```

This is important because condition-variable waits can wake up spuriously. The predicate form ensures the thread continues only when the condition is actually true.

35.5.6 Waiting with a timeout

```
#include <condition_variable>
#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex m;
std::condition_variable cv;
bool ready = false;

void consumer()
{
    std::unique_lock<std::mutex> lock(m);

    bool ok = cv.wait_for(lock, std::chrono::milliseconds(500), []{
        return ready;
    });
```

```
    if (ok)
    {
        std::cout << "Condition satisfied\n";
    }
    else
    {
        std::cout << "Timed out\n";
    }
}

int main()
{
    std::thread t(consumer);
    t.join();
}
```

35.5.7 When to use `notify_one` vs `notify_all`

Use:

- `notify_one()` when one waiting thread should wake
- `notify_all()` when all waiting threads should wake and recheck their predicates

35.6 `std::atomic`

`std::atomic` provides atomic operations on objects shared between threads. It is defined in:

```
#include <atomic>
```

An atomic object allows certain operations to occur safely as indivisible inter-thread operations, often without using a mutex.

35.6.1 Why atomics exist

For some shared state, using a mutex is more than necessary. A simple counter, flag, or state variable may be better represented as an atomic object.
Examples of suitable atomic use cases:

- counters
- flags
- simple state transitions
- lock-free coordination primitives

35.6.2 Basic atomic counter

```
#include <atomic>
#include <iostream>
#include <thread>

std::atomic<int> counter{0};

void increment()
{
    for (int i = 0; i < 10000; ++i)
    {
        ++counter;
    }
}

int main()
{
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << counter << '\n';
}
```

This is safe because the counter is atomic.

35.6.3 Load and store

The basic operations are reading and writing.

```
#include <atomic>
#include <iostream>

int main()
{
    std::atomic<int> value{10};

    int x = value.load();
    value.store(25);

    std::cout << x << ' ' << value.load() << '\n';
}
```

35.6.4 Atomic flag-style example

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>
```

```
std::atomic<bool> ready{false};

void worker()
{
    while (!ready.load())
    {
    }

    std::cout << "Worker sees ready = true\n";
}

int main()
{
    std::thread t(worker);

    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    ready.store(true);

    t.join();
}
```

This is much safer than using a plain shared `bool`.

35.6.5 Read-modify-write operations

Atomic types support operations such as:

- increment and decrement
- `fetch_add()`
- `fetch_sub()`
- compare-and-exchange

```
#include <atomic>
#include <iostream>

int main()
{
    std::atomic<int> value{5};

    int old = value.fetch_add(3);

    std::cout << "old = " << old << '\n';
    std::cout << "new = " << value.load() << '\n';
}
```

35.6.6 Compare-and-exchange

Compare-and-exchange is one of the most important atomic operations.

```cpp
#include <atomic>
#include <iostream>

int main()
{
    std::atomic<int> value{10};
    int expected = 10;

    bool changed = value.compare_exchange_strong(expected, 20);

    std::cout << std::boolalpha << changed << '\n';
    std::cout << value.load() << '\n';
}
```

This operation updates the atomic only if it currently matches the expected value.

35.6.7 Memory ordering

Atomic operations support memory-order parameters. By default, many simple atomic operations use sequentially consistent ordering.
For beginners, the safest initial rule is:

- start with default atomic operations
- learn relaxed and other memory orders only after the basics are fully understood

Memory ordering is powerful but advanced. It should not be used casually.

35.6.8 When not to use atomics

Atomics are not a general replacement for mutexes. They are best for simple shared state. If the shared invariant involves multiple related variables or complex logic, a mutex is often clearer and safer.
Bad instinct:

> Replace every mutex with an atomic.

Better instinct:

> Use atomics for simple atomic state. Use mutexes when protecting larger invariants or compound state.

35.7 Practical comparisons

35.7.1 `std::thread` vs `std::jthread`

A practical summary is:

- `std::thread` is the basic thread type
- `std::jthread` is often safer and more modern in C++20 code
- `std::jthread` integrates naturally with cooperative stop requests

35.7.2 `std::mutex` vs `std::atomic`

Use a mutex when:

- multiple variables form one invariant
- the critical section is more than one simple atomic update
- code clarity is more important than low-level lock-free style

Use an atomic when:

- the shared state is simple
- a single variable or simple read-modify-write operation is enough
- you truly need atomic semantics rather than a protected critical section

35.7.3 `std::lock_guard` vs `std::scoped_lock`

For one mutex, both are useful. `std::scoped_lock` becomes especially attractive when multiple mutexes must be locked together.

35.7.4 `std::condition_variable` vs busy waiting

Condition variables are preferred when one thread must wait for a state change. Busy waiting wastes CPU and is usually poorer design.

35.8 Windows-oriented practical examples

35.8.1 Example 1: background worker with `std::jthread`

```
#include <chrono>
#include <iostream>
#include <stop_token>
#include <thread>

void background_logger(std::stop_token st)
{
    while (!st.stop_requested())
    {
        std::cout << "Background logger tick\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
    }
}

int main()
{
    std::jthread worker(background_logger);
    std::this_thread::sleep_for(std::chrono::milliseconds(600));
    worker.request_stop();
}
```

35.8.2 Example 2: queue-ready notification

```
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <queue>
#include <thread>

std::mutex m;
std::condition_variable cv;
std::queue<int> q;
bool done = false;

void producer()
{
    {
        std::lock_guard<std::mutex> lock(m);
        q.push(42);
        done = true;
    }
    cv.notify_one();
}

void consumer()
{
    std::unique_lock<std::mutex> lock(m);
    cv.wait(lock, []{ return !q.empty() || done; });

    if (!q.empty())
    {
        std::cout << q.front() << '\n';
        q.pop();
    }
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);

    t1.join();
    t2.join();
}
```

35.8.3 Example 3: atomic status flag

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>

std::atomic<bool> running{true};
```

```
void worker()
{
    while (running.load())
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    std::cout << "Worker stopped\n";
}

int main()
{
    std::thread t(worker);

    std::this_thread::sleep_for(std::chrono::milliseconds(300));
    running.store(false);

    t.join();
}
```

35.9 Best practices

35.9.1 Prefer RAII for synchronization

Use RAII wrappers such as:

- `std::scoped_lock`
- `std::lock_guard`
- `std::unique_lock`

Avoid manual `lock()` and `unlock()` unless there is a very good reason.

35.9.2 Prefer `std::jthread` when available

In C++20 code, `std::jthread` is often a safer default than `std::thread` for scoped thread ownership.

35.9.3 Use condition variables instead of sleeping loops

Do not write polling loops with arbitrary sleeps when proper waiting semantics are required.

35.9.4 Use atomics only for suitably simple shared state

Atomics are powerful but do not replace all synchronization patterns.

35.9.5 Protect invariants, not just variables

If several values must change together consistently, a mutex-protected critical section is often the right solution.

Summary

Modern C++ provides a solid set of standard concurrency tools:

- `std::thread` for basic thread creation and management
- `std::jthread` for safer C++20 thread ownership and cooperative stop handling
- `std::mutex` for mutual exclusion
- `std::scoped_lock` for RAII locking, especially across multiple mutexes
- `std::condition_variable` for waiting on state changes without busy polling
- `std::atomic` for lock-free atomic operations on simple shared state

The most important design lesson is that concurrency tools are not interchangeable decorations. Each exists for a specific category of problem. Strong modern C++ code chooses the tool that matches the real synchronization need, keeps shared mutable state as small as possible, and relies on RAII to make correct behavior easier to maintain.

Modern Concurrency

36.1 Cancellation

Modern C++ (since C++20) introduces a standardized and structured mechanism for cooperative thread cancellation using the `<stop_token>` facilities. Unlike older techniques based on shared flags, this approach provides a well-defined and thread-safe model for requesting termination of asynchronous operations.

36.1.1 Cooperative cancellation model

Cancellation in modern C++ is **cooperative**, meaning:

- threads are not forcibly stopped
- a stop request is issued
- the running thread periodically checks whether it should stop
- the thread exits gracefully when appropriate

This avoids unsafe termination, resource leaks, and undefined behavior.

36.1.2 Core components

The cancellation system consists of three key types:

- `std::stop_source` issues a stop request
- `std::stop_token` observes the stop request
- `std::stop_callback` executes logic when cancellation occurs

The `std::stop_token` provides functions such as:

- `stop_requested()`
- `stop_possible()`

A stop request is communicated through a shared stop-state, and querying or requesting stop is safe and does not introduce data races.

36.1.3 Cancellation with `std::jthread`

`std::jthread` integrates cancellation directly. When a thread function accepts a `std::stop_token`, it is automatically provided.

```
#include <chrono>
#include <iostream>
#include <thread>

void worker(std::stop_token st)
{
    while (!st.stop_requested())
    {
        std::cout << "Working...\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    std::cout << "Stopping gracefully\n";
}

int main()
{
    std::jthread t(worker);

    std::this_thread::sleep_for(std::chrono::milliseconds(300));
    t.request_stop();
}
```

A stop request is issued atomically and becomes visible to all associated tokens.

36.1.4 Cancellation with condition variables

Modern C++ also supports interruptible waits using `std::condition_variable_any` with stop tokens, allowing threads to be woken up immediately when cancellation is requested.

```
#include <condition_variable>
#include <mutex>
#include <thread>
#include <stop_token>

std::mutex m;
std::condition_variable_any cv;

void waiter(std::stop_token st)
{
    std::unique_lock lock(m);
    cv.wait(lock, st, [] { return false; });
}
```

36.1.5 Why cooperative cancellation is important

This model ensures:

- predictable cleanup
- safe resource release (RAII)
- no forced thread termination
- composable cancellation across APIs

36.2 Structured threading

Structured threading is a design principle where thread lifetimes are tied to program scopes. Instead of unmanaged background threads, threads are treated as scoped resources.

36.2.1 The problem with unstructured threads

Traditional thread usage:

- threads may outlive their owning objects
- forgetting `join()` leads to program termination
- difficult reasoning about ownership and lifetime

36.2.2 Structured approach with `std::jthread`

`std::jthread` enforces structured lifetime:

- automatically joins in its destructor
- automatically requests stop before joining
- integrates with cancellation

This eliminates a major class of bugs where threads remain joinable at destruction.

```
#include <thread>
#include <iostream>

int main()
{
    {
        std::jthread t([] {
            std::cout << "Scoped thread\n";
        });
    } // automatically joined here
}
```

36.2.3 Structured concurrency principles

Key ideas:

- threads should not escape their logical scope
- lifetime must be bounded and predictable
- parent scope owns child threads
- cancellation propagates downward

36.2.4 Comparison with manual thread management

```
void unsafe()
{
    std::thread t([]{});
    // exception here => terminate
    t.join();
}
```

```
void safe()
{
    std::jthread t([]{});
    // exception here is safe
}
```

Structured threading improves exception safety and program correctness.

36.3 Memory model basics

The C++ memory model defines how operations on memory behave across threads. It specifies when changes made by one thread become visible to another.

36.3.1 Data race definition

A data race occurs when:

- two or more threads access the same memory location
- at least one access is a write
- no synchronization establishes ordering

A data race results in undefined behavior.

36.3.2 Happens-before relationship

Correct concurrent programs rely on the **happens-before** relationship.
If operation A happens-before operation B:

- all effects of A are visible to B
- execution is well-defined

Synchronization primitives such as mutexes and atomics establish happens-before relationships.

36.3.3 Atomic operations

Atomic operations guarantee:

- no data races
- indivisible read-modify-write
- ordering guarantees depending on memory order

36.3.4 Memory ordering levels

The main ordering levels include:

- sequential consistency (default, strongest)
- acquire/release
- relaxed ordering

For most applications, sequential consistency is preferred initially because it is easiest to reason about.

36.3.5 Example: message passing

```
#include <atomic>
#include <thread>
#include <cassert>

int data = 0;
std::atomic<bool> ready{false};

void producer()
{
    data = 42;
    ready.store(true, std::memory_order_release);
}

void consumer()
{
    while (!ready.load(std::memory_order_acquire))
    {
```

```
    }
    assert(data == 42);
}
```

This ensures:

- write to `data` becomes visible
- ordering is enforced via acquire/release

36.3.6 Synchronization primitives

The following establish ordering:

- mutex lock/unlock
- condition variable wait/notify
- atomic operations with ordering

36.3.7 Key rule

Without synchronization, visibility between threads is not guaranteed.

36.4 Common pitfalls

Modern concurrency introduces subtle and dangerous errors. Understanding these pitfalls is essential.

36.4.1 Forgetting to join threads

```
std::thread t([]{});
```

If the thread object is destroyed while still joinable, the program calls `std::terminate()`.
Solution:

- always call `join()`
- or use `std::jthread`

36.4.2 Detached threads and lifetime issues

Detached threads may access objects that no longer exist.

```
void dangerous()
{
    int x = 10;
    std::thread([&] { /* uses x */ }).detach();
}
```

This leads to undefined behavior due to dangling references.

36.4.3 Data races

```
int counter = 0;
```

Multiple threads modifying this without synchronization create undefined behavior.

36.4.4 Busy waiting

```
while (!ready)
{
}
```

Problems:

- wastes CPU
- may never observe updates without atomic or synchronization

Solution:

- use condition variables
- or atomic wait/notify

36.4.5 Incorrect use of atomics

Using atomics incorrectly can still lead to logic errors.

- missing ordering constraints
- incorrect assumptions about visibility

36.4.6 Deadlocks

Deadlocks occur when threads wait on each other indefinitely.

```
std::mutex m1, m2;

void f()
{
    std::lock_guard lock1(m1);
    std::lock_guard lock2(m2);
}

void g()
{
    std::lock_guard lock1(m2);
    std::lock_guard lock2(m1);
}
```

Solution:

- use consistent lock ordering
- or use `std::scoped_lock`

36.4.7 Ignoring cancellation

Threads that do not check cancellation signals may:

- run indefinitely
- block shutdown
- leak resources

36.4.8 Blocking operations without interruption

Threads waiting on blocking operations without cancellation support cannot respond to stop requests.
Solution:

- use interruptible waits
- integrate stop tokens into waiting logic

36.4.9 Overusing threads

Creating too many threads can degrade performance due to:

- context switching
- memory overhead
- scheduling contention

Better approach:

- use thread pools
- reuse threads

Summary

Modern C++ concurrency emphasizes safety, structure, and correctness.

- cancellation is cooperative and standardized using stop tokens
- structured threading ties thread lifetime to scope and improves safety
- the memory model defines visibility and ordering between threads
- data races lead to undefined behavior and must be avoided
- common pitfalls include lifetime errors, deadlocks, and misuse of atomics

The most important design principle is:

> Write concurrent code that is simple, structured, and explicitly synchronized, rather than clever or implicit.

Modern C++ provides the tools, but correctness depends on disciplined design.

Part XI

Modern Features (C++20 / C++23 / C++26)

Ranges

37.1 Motivation

The Ranges library, introduced in C++20 and extended in C++23, represents a fundamental redesign of how algorithms interact with data. Instead of relying on pairs of iterators, ranges provide a higher-level abstraction that models sequences directly.
A **range** is any object that provides `begin()` and `end()`, allowing iteration over its elements. This abstraction generalizes containers, arrays, and even generated sequences into a unified model.
The primary motivations behind ranges are:

37.1.1 Eliminating Iterator Complexity

Traditional STL algorithms operate on iterator pairs:

```
std::sort(vec.begin(), vec.end());
```

This approach is error-prone:

- Passing mismatched iterators
- Off-by-one errors
- Reduced readability

Ranges eliminate this by operating directly on the range:

```
std::ranges::sort(vec);
```

37.1.2 Composability

The ranges library is designed to be composable. Instead of writing multiple loops or intermediate containers, operations can be chained together into a single expression.

37.1.3 Lazy Evaluation

A key design principle is laziness. Range adaptors do not process data immediately. Instead, computation is deferred until iteration occurs. This leads to:

- Reduced memory usage
- Fewer temporary allocations
- Improved performance

37.1.4 Safer and More Expressive Code

Ranges reduce boilerplate and make intent clearer. They are less error-prone because:

- Types are constrained using concepts
- Operations are applied only when valid
- The structure of data transformations is explicit

37.2 Views

A **view** is a lightweight, non-owning abstraction over a range. It does not store data but provides a way to access or transform it. Views are central to the ranges design.

37.2.1 Properties of Views

- Do not own elements
- Cheap to copy and move
- Lazily evaluated
- Composable with other views

37.2.2 Basic Example

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

int main()
{
    std::vector<int> v {1, 2, 3, 4, 5};

    auto even = std::views::filter(v, [](int x) {
        return x % 2 == 0;
    });

    for (int x : even)
        std::cout << x << " ";
}
```

37.2.3 Common View Adaptors

- `std::views::filter` selects elements based on predicate
- `std::views::transform` applies transformation
- `std::views::take` takes first N elements

- `std::views::drop` skips first N elements
- `std::views::iota` generates sequence
- `std::views::reverse` reverses order
- `std::views::split` splits ranges

37.2.4 Example with Multiple Views

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

int main()
{
    std::vector<int> data {1, 2, 3, 4, 5, 6};

    auto result = std::views::filter(data, [](int x) {
                        return x % 2 == 0;
                  })
                | std::views::transform([](int x) {
                        return x * x;
                  });

    for (int x : result)
        std::cout << x << " ";
}
```

37.3 Pipelines

Pipelines are one of the most powerful features of ranges. They allow chaining operations using the pipe operator `|`.

37.3.1 Concept of Pipelines

A pipeline is a sequence of transformations applied to a range:

```
range | view1 | view2 | view3
```

Each stage produces another view, forming a transformation chain.

37.3.2 Lazy Execution Model

Each operation in the pipeline is not executed immediately. Instead:

- The pipeline defines a transformation
- Execution happens only during iteration

This means a pipeline runs in a single pass without creating intermediate containers.

37.3.3 Example Pipeline

```
#include <iostream>
#include <ranges>

int main()
{
    auto pipeline =
        std::views::iota(0, 20)
        | std::views::filter([](int x) { return x % 2 == 0; })
        | std::views::transform([](int x) { return x * x; })
        | std::views::take(5);

    for (int x : pipeline)
        std::cout << x << " ";
}
```

37.3.4 Mixing with Algorithms

Pipelines can be combined with range algorithms:

```
#include <vector>
#include <ranges>
#include <algorithm>

int main()
{
    std::vector<int> v {5, 3, 8, 1, 9};

    auto filtered = v | std::views::filter([](int x) {
        return x > 3;
    });

    std::ranges::sort(v);
}
```

37.3.5 Materializing Results (C++23)

In C++23, `std::ranges::to` allows converting a pipeline result into a container:

```
#include <vector>
#include <ranges>

int main()
{
    std::vector<int> v {1,2,3,4,5,6};

    auto result =
        v | std::views::filter([](int x){ return x % 2 == 0; })
          | std::ranges::to<std::vector>();

    for (int x : result)
```

```
        std::cout << x << " ";
}
```

37.4 Cleaner Code Patterns

Ranges enable writing cleaner, more declarative code compared to traditional STL patterns.

37.4.1 Before Ranges (Imperative Style)

```
std::vector<int> result;

for (int x : data)
{
    if (x % 2 == 0)
        result.push_back(x * x);
}
```

37.4.2 After Ranges (Declarative Style)

```
auto result =
    data
    | std::views::filter([](int x){ return x % 2 == 0; })
    | std::views::transform([](int x){ return x * x; });
```

37.4.3 Avoiding Temporary Containers

Traditional approach:

```
std::vector<int> temp;

for (int x : data)
    if (x > 10)
        temp.push_back(x);

std::vector<int> result;

for (int x : temp)
    result.push_back(x * 2);
```

Ranges approach:

```
auto result =
    data
    | std::views::filter([](int x){ return x > 10; })
    | std::views::transform([](int x){ return x * 2; });
```

37.4.4 Working with Structured Data

```
#include <vector>
#include <tuple>
#include <ranges>
```

```cpp
#include <iostream>

int main()
{
    std::vector<std::tuple<int, std::string>> data {
        {1, "A"}, {2, "B"}, {3, "C"}
    };

    for (const auto& name : data | std::views::elements<1>)
        std::cout << name << " ";
}
```

37.4.5 Sliding Window (C++23)

```cpp
#include <vector>
#include <ranges>
#include <iostream>

int main()
{
    std::vector<int> v {1,2,3,4,5};

    for (auto window : v | std::views::slide(3))
    {
        for (int x : window)
            std::cout << x << " ";
        std::cout << "\n";
    }
}
```

37.4.6 Guidelines for Cleaner Code

- Prefer views over manual loops
- Avoid intermediate containers unless necessary
- Use pipelines to express transformations clearly
- Keep lambdas simple and focused
- Use `auto` for complex range types

37.4.7 When Not to Use Ranges

- Extremely performance-critical inner loops (measure first)
- Very simple operations where ranges add complexity
- Legacy codebases without C++20 support

Summary

The Ranges library transforms how modern C++ handles data processing:

- Provides safer abstraction over iterators
- Enables lazy, composable transformations
- Eliminates boilerplate and intermediate storage
- Improves readability and maintainability

Ranges represent a shift toward a more functional and declarative programming style in C++, while maintaining zero-cost abstractions and high performance.

Compile-Time Programming

38.1 `constexpr`

Compile-time programming in Modern C++ allows part of a program to be evaluated during translation rather than at runtime. The most widely used language tool for this purpose is `constexpr`. A `constexpr` entity declares that it may participate in constant evaluation when the surrounding rules allow it.
In practical terms, `constexpr` serves two closely related goals:

- expressing that a value may be known at compile time,
- allowing functions and objects to be used in constant-expression contexts.

A beginner should understand an important rule immediately: `constexpr` does not mean that evaluation always happens at compile time. It means that compile-time evaluation is permitted when all semantic requirements are satisfied. The same function may still run at runtime when called with runtime data.

38.1.1 Why `constexpr` matters

Before Modern C++, compile-time computation was possible, but often through difficult template metaprogramming techniques. `constexpr` made compile-time logic more readable, more direct, and closer to ordinary code.
It is useful for:

- constants that must be available during translation,
- array bounds and template arguments,
- lookup tables,
- validation logic that should fail early,
- performance-sensitive code where some work can be shifted from runtime to compile time,
- designing libraries with stronger guarantees.

38.1.2 `constexpr` variables

A `constexpr` variable must be initialized with a constant expression.

```
#include <array>

constexpr int buffer_size = 256;
std::array<int, buffer_size> data{};
```

Here, `buffer_size` is known during translation, so it can be used where the language requires a constant expression.

38.1.3 `constexpr` functions

A `constexpr` function may be evaluated at compile time if called with arguments that are themselves usable in constant evaluation.

```
constexpr int square(int x)
{
    return x * x;
}

constexpr int value = square(12);
```

The same function may also execute at runtime:

```
#include <iostream>

constexpr int square(int x)
{
    return x * x;
}

int main()
{
    int n = 0;
    std::cin >> n;
    std::cout << square(n) << '\n';
}
```

This dual-use nature is one of the greatest strengths of `constexpr`: one function can serve both compile-time and runtime callers.

38.1.4 A more realistic example

```
#include <array>
#include <cstddef>

constexpr int sum_array(const std::array<int, 5>& arr)
{
    int total = 0;
    for (std::size_t i = 0; i < arr.size(); ++i)
        total += arr[i];
    return total;
}

constexpr std::array<int, 5> values{1, 2, 3, 4, 5};
constexpr int result = sum_array(values);
```

This style is much easier to read than older template-heavy approaches.

38.1.5 Compile-time branching

Compile-time programming often becomes more powerful when combined with `if constexpr`.

```
#include <type_traits>
#include <iostream>

template<typename T>
constexpr T process(T value)
{
    if constexpr (std::is_integral_v<T>)
        return value * 2;
    else
        return value + static_cast<T>(0.5);
}

int main()
{
    std::cout << process(10) << '\n';
    std::cout << process(3.0) << '\n';
}
```

`if constexpr` removes invalid branches during compilation, which makes generic code cleaner and safer.

38.1.6 `constexpr` constructors and objects

User-defined types can also participate in compile-time computation.

```
class Point
{
public:
    constexpr Point(int x, int y) : x_(x), y_(y) {}

    constexpr int x() const { return x_; }
    constexpr int y() const { return y_; }

private:
    int x_;
    int y_;
};

constexpr Point p(10, 20);
constexpr int px = p.x();
```

This is extremely useful for small value types, math utilities, coordinate systems, dimensions, and strongly typed wrappers.

38.1.7 Using `std::is_constant_evaluated`

Sometimes a function needs to behave differently depending on whether it is being evaluated at compile time or runtime.

```cpp
#include <type_traits>
#include <iostream>

constexpr int compute(int x)
{
    if (std::is_constant_evaluated())
        return x * x;
    else
        return x + x;
}

constexpr int a = compute(5);

int main()
{
    int n = 5;
    std::cout << compute(n) << '\n';
}
```

This technique should be used carefully. It is powerful, but a library becomes harder to understand if compile-time and runtime behavior diverge too much.

38.1.8 A compile-time lookup table

```cpp
#include <array>
#include <cstddef>

constexpr std::array<int, 10> make_squares()
{
    std::array<int, 10> arr{};
    for (std::size_t i = 0; i < arr.size(); ++i)
        arr[i] = static_cast<int>(i * i);
    return arr;
}

constexpr auto squares = make_squares();
```

This pattern is excellent when the data is deterministic and reused many times.

38.1.9 Guidelines for `constexpr`

- Mark a function `constexpr` when compile-time use is meaningful and natural.
- Do not force `constexpr` on every function. Use it where it improves design.
- Prefer ordinary readable code over complicated compile-time tricks.
- Use `constexpr` for value types, utilities, parsers of simple literals, validation helpers, and static data generation.
- Keep compile-time work reasonable. Heavy compile-time computation increases build times.

38.2 `consteval`

If `constexpr` means “may be evaluated at compile time,” then `consteval` means “must be evaluated at compile time” for every potentially evaluated call.
A `consteval` function is called an **immediate function**. It is stricter than `constexpr` and is used when runtime execution would be conceptually wrong or unsafe for the intended design.

38.2.1 Why `consteval` exists

Sometimes a library author does not merely want to *allow* compile-time evaluation. They want to *require* it.
This is useful when:

- input must be validated during translation,
- a function is meaningful only for constant arguments,
- generated values must never depend on runtime state,
- incorrect use should fail immediately at compile time.

38.2.2 Basic example

```
consteval int cube(int x)
{
    return x * x * x;
}

constexpr int a = cube(3);
```

This is valid, because the call is a compile-time call.
The following would be invalid:

```
#include <iostream>

consteval int cube(int x)
{
    return x * x * x;
}

int main()
{
    int n = 4;
    // int r = cube(n); // error: call is not a constant expression
}
```

38.2.3 Immediate validation

A strong use case is validation of configuration or literal-based input.

```
consteval int checked_port(int port)
{
    if (port < 1 || port > 65535)
        throw "invalid port";
    return port;
}

constexpr int server_port = checked_port(8080);
```

Here, an invalid port becomes a compile-time error rather than a late runtime failure.

38.2.4 Generating compile-time IDs

```
#include <cstddef>

consteval std::size_t str_length(const char* s)
{
    std::size_t n = 0;
    while (s[n] != '\0')
        ++n;
    return n;
}

constexpr std::size_t len = str_length("CompileTime");
```

This style is useful in metaprogramming, fixed-size buffers, compile-time parsing, and embedded configuration systems.

38.2.5 `consteval` versus `constexpr`

The difference is conceptual and practical:

- `constexpr`: compile-time evaluation is allowed.
- `consteval`: compile-time evaluation is required.

A `constexpr` function can be called with runtime values. A `consteval` function cannot.

38.2.6 Factory-style compile-time creation

```
struct Config
{
    int width;
    int height;
};

consteval Config make_config(int w, int h)
{
    if (w <= 0 || h <= 0)
        throw "dimensions must be positive";
    return {w, h};
}

constexpr Config cfg = make_config(800, 600);
```

This is especially attractive when incorrect data must never reach runtime.

38.2.7 Practical limitations

`consteval` is not a general replacement for `constexpr`. It is best reserved for APIs that are explicitly compile-time-only. Overusing it can make ordinary program construction unnecessarily rigid. If a function could legitimately be useful at runtime, `constexpr` is often the better choice.

38.2.8 Windows-oriented note

When compiling on Windows, immediate functions are well-suited for:

- compile-time configuration of fixed protocol values,
- validation of static dimensions and limits,
- generation of small lookup structures,
- checking format assumptions in systems code before the program is linked.

38.3 `constinit`

`constinit` solves a different problem. It does not make a variable constant, and it does not mean compile-time usage in the same way as `constexpr`. Instead, it guarantees that an object with static or thread storage duration is initialized by constant initialization.
This is mainly about **initialization timing and safety**.

38.3.1 The problem it addresses

Global and namespace-scope objects can suffer from initialization-order issues. In large systems, especially multi-file projects, one global object may depend on another object in a different translation unit. When initialization happens dynamically, the order can become a source of subtle bugs.
`constinit` helps ensure that initialization is performed in the constant-initialization phase rather than delayed dynamic initialization.

38.3.2 Basic example

```
constinit int global_counter = 0;
```

This guarantees that the object is initialized statically.

38.3.3 `constinit` is not `const`

The following is valid:

```
constinit int value = 10;

int main()
{
    value = 20;
}
```

This is an important distinction:

- `constexpr` typically expresses compile-time constant usability,
- `constinit` expresses guaranteed static initialization,
- `const` expresses immutability through that name.

38.3.4 Useful for global state that must start safely

```
#include <atomic>

constinit std::atomic<int> active_sessions{0};

int main()
{
    active_sessions.fetch_add(1);
}
```

This pattern is useful in backend, systems, and multithreaded programs where a global object must be ready before dynamic startup logic begins.

38.3.5 Thread-local example

```
thread_local constinit int tls_id = 0;
```

This states that the thread-local object must also have constant initialization.

38.3.6 A practical multi-file design idea

Suppose a project has a logging subsystem, metrics subsystem, and network subsystem. If some global counters or fixed configuration values exist at namespace scope, `constinit` can protect the program from accidental dynamic initialization of those objects.

```
// globals.hpp
#pragma once

extern constinit int default_timeout_ms;
```

```
// globals.cpp
constinit int default_timeout_ms = 5000;
```

The object is initialized statically, but remains mutable if the design requires it.

38.3.7 When to prefer `constinit`

Use `constinit` when:

- an object has static or thread storage duration,
- constant initialization is required,
- immutability is not required,
- you want to prevent the static initialization order problem from becoming worse.

38.3.8 When `constexpr` is better

Use `constexpr` instead when:

- the object should be a constant expression,
- the value is conceptually immutable,
- the value must be usable in contexts requiring constant expressions,
- the object is a true compile-time value rather than merely safely initialized static storage.

38.3.9 Common confusion

The three keywords solve different design problems:

- `constexpr`: compile-time-capable value or function.
- `consteval`: compile-time-only function.
- `constinit`: static-initialization guarantee for static or thread storage objects.

38.4 Design implications

Compile-time programming is not only about speed. It changes the architecture of libraries and applications.

38.4.1 Earlier error detection

One of the strongest design benefits is moving failure from runtime to compile time.
For example, invalid dimensions, unsupported options, malformed literal-based input, and impossible combinations can be rejected before the program ever runs.

```
struct Dimensions
{
    int width;
    int height;
};

consteval Dimensions make_dimensions(int w, int h)
```

```
{
    if (w <= 0 || h <= 0)
        throw "invalid dimensions";
    return {w, h};
}

constexpr auto screen = make_dimensions(1920, 1080);
```

This style produces stronger contracts and clearer APIs.

38.4.2 Library interface clarity

A public API that uses `constexpr`, `consteval`, and `constinit` communicates design intent explicitly.

- `constexpr` says: this operation can participate in constant evaluation.
- `consteval` says: this operation exists only for compile-time use.
- `constinit` says: this static object must be initialized safely and early.

These are not just keywords; they are part of the library contract.

38.4.3 Cleaner metaprogramming

Older metaprogramming techniques often relied heavily on templates, specializations, and recursive type tricks. Modern compile-time programming allows many of those tasks to be expressed with ordinary functions and straightforward control flow.

```
constexpr bool is_power_of_two(unsigned int x)
{
    return x != 0 && (x & (x - 1)) == 0;
}

static_assert(is_power_of_two(64));
```

This is easier to teach, easier to debug, and easier to maintain.

38.4.4 Safer global design

Global initialization is traditionally one of the danger zones in C++. `constinit` improves robustness in large codebases, especially where several translation units participate in system startup.
In Windows applications, services, backend servers, and libraries, this matters because startup order bugs can remain hidden for long periods and appear only in certain build configurations.

38.4.5 Build-time cost versus runtime cost

Compile-time programming shifts work from program execution to translation. This can improve runtime efficiency, but it may increase build times.
A mature design therefore balances:

- runtime performance,

- readability,
- compile-time cost,
- error quality,
- maintainability.

Not every calculation should be pushed into the compiler. Compute at compile time when it improves guarantees, safety, or essential performance characteristics.

38.4.6 Better invariants

Compile-time techniques are ideal for expressing invariants that should never be violated.

```
template<int N>
struct Buffer
{
    static_assert(N > 0, "Buffer size must be positive");
    int data[N]{};
};
```

Such designs make illegal states harder to represent.

38.4.7 Stronger domain-specific abstractions

Compile-time programming is especially powerful for:

- units and dimensions,
- fixed protocols,
- parsers for restricted literals,
- validated configuration values,
- embedded lookup data,
- numerical policies,
- type-safe wrappers around low-level constants.

38.4.8 A combined example

The following example shows all three ideas in one design:

```
#include <array>
#include <atomic>
#include <cstddef>

consteval int checked_capacity(int n)
{
    if (n <= 0 || n > 1024)
```

```
        throw "capacity out of range";
    return n;
}

template<typename T, int N>
class FixedBuffer
{
public:
    constexpr FixedBuffer() = default;

    constexpr void set(std::size_t index, const T& value)
    {
        data_[index] = value;
    }

    constexpr const T& get(std::size_t index) const
    {
        return data_[index];
    }

    constexpr std::size_t size() const
    {
        return N;
    }

private:
    std::array<T, N> data_{};
};

constinit std::atomic<int> live_objects{0};

int main()
{
    constexpr int cap = checked_capacity(16);
    FixedBuffer<int, cap> buf{};

    buf.set(0, 42);
    live_objects.fetch_add(1);

    return buf.get(0);
}
```

In this design:

- `checked_capacity()` is `consteval`, so invalid capacities fail during translation,
- `FixedBuffer` uses `constexpr` member functions so objects can participate in compile-time evaluation where appropriate,
- `live_objects` uses `constinit` to guarantee safe static initialization.

38.4.9 Windows compilation examples

The following examples are typical command lines for Windows environments.

38.4.10 MSVC

```
cl /std:c++20 /EHsc /W4 main.cpp
```

For newer experimental language work in modern compiler modes:

```
cl /std:c++latest /EHsc /W4 main.cpp
```

38.4.11 Clang on Windows

```
clang++ -std=c++20 -Wall -Wextra -pedantic main.cpp -o app.exe
```

For post-C++23 work:

```
clang++ -std=c++2c -Wall -Wextra -pedantic main.cpp -o app.exe
```

38.4.12 GCC through MinGW on Windows

```
g++ -std=c++20 -Wall -Wextra -pedantic main.cpp -o app.exe
```

When supported by the installed toolchain:

```
g++ -std=c++2c -Wall -Wextra -pedantic main.cpp -o app.exe
```

38.4.13 Best practices

- Use `constexpr` for values and algorithms that naturally benefit from compile-time availability.
- Use `consteval` for immediate validation and compile-time-only factories.
- Use `constinit` for static and thread-local objects whose initialization timing must be guaranteed.
- Prefer readable compile-time code over clever template tricks.
- Measure compile-time cost in large projects.
- Do not confuse immutability, constant evaluation, and static initialization. They are related but different concepts.

38.4.14 Common mistakes

- Assuming `constexpr` always means compile-time execution.
- Using `consteval` where runtime use would actually be useful.
- Confusing `constinit` with `const`.
- Moving too much logic into compile time without architectural benefit.
- Writing compile-time code that is technically valid but too hard for future maintainers to understand.

Summary

Compile-time programming in Modern C++ has evolved into a practical and expressive design discipline.
`constexpr` enables functions and values to participate in constant evaluation when possible. `consteval` enforces compile-time-only evaluation for immediate functions. `constinit` guarantees constant initialization of static and thread-local objects without implying immutability.
Together, these tools let C++ programmers design APIs that detect errors earlier, encode stronger invariants, reduce certain classes of startup bugs, and express domain rules directly in the type system and in translation-time logic.
Well-used compile-time programming does not merely optimize execution. It improves program correctness, design clarity, and long-term maintainability.

Modules

39.1 Motivation

C++ modules were introduced to solve long-standing structural problems in the traditional header-based model. For decades, C++ programs have been organized around source files and header files, where declarations are repeatedly included into many translation units through the preprocessor. This approach works, but it has several well-known technical costs:

- repeated parsing of the same headers in many translation units,
- heavy dependence on textual inclusion,
- macro leakage across file boundaries,
- fragile build scaling in large systems,
- poor isolation between library boundaries,
- long compile times in projects with large dependency graphs.

Modules provide a language-level alternative. Instead of copying declarations textually with `#include`, a module is compiled once as a separate language unit and then imported by other translation units with `import`. This changes the compilation model from textual inclusion to semantic import.
A practical way to understand the motivation is to compare two mental models.
With headers, the compiler effectively sees this idea:

```
// source file
#include "library.hpp"
#include "other.hpp"
#include "more.hpp"
```

Each `#include` inserts text into the current translation unit. The preprocessor runs first, macros expand, include guards attempt to reduce duplication, and the compiler repeatedly parses declarations that may already have been parsed elsewhere in the project.
With modules, the intent is different:

```
import math;
import network;
import storage;
```

Here the compiler imports already-compiled module information rather than copying source text into the importing file.

39.1.1 Why the header model became painful

The historical header model is tightly coupled to the C preprocessor. This creates several recurring issues:

- Macros can silently affect included code.
- Include order can matter.
- Headers often require careful forward declarations and include minimization.
- One change in a widely included header can trigger large rebuilds.
- Internal implementation details can accidentally leak through headers.

In small programs, these issues may feel manageable. In large industrial systems, they become architectural problems.

39.1.2 What modules try to improve

Modules aim to improve C++ program structure in several ways:

- better separation between interface and implementation,
- less dependence on preprocessor-based inclusion,
- improved compile-time scalability,
- stronger encapsulation,
- clearer dependency boundaries,
- fewer accidental transitive exposure problems.

This is one of the most important language-level modernization steps in C++20.

39.1.3 Named modules and header units

In standard C++ module terminology, there are two important forms:

- **Named modules**
- **Header units**

Named modules are the main architectural feature and are intended for modern library design. Header units provide a bridge for importing some headers through the modules machinery.
A named module typically begins with:

```
export module math;
```

A user then imports it with:

```
import math;
```

This model is fundamentally different from textual inclusion.

39.1.4 A first minimal example

The following example shows a small module interface and its use.

39.1.5 Module interface unit

```
// math.ixx
export module math;

export int add(int a, int b)
{
    return a + b;
}
```

39.1.6 Importing translation unit

```
// main.cpp
import math;
#include <iostream>

int main()
{
    std::cout << add(10, 20) << '\n';
}
```

This already demonstrates the core idea: the consumer imports a module name rather than including a header file.

39.1.7 Why this matters for large projects

In a large codebase, modules can support better layering. Instead of exporting large collections of transitive headers, a library can define a narrower semantic interface. This often results in:

- more deliberate public APIs,
- fewer unintended dependencies,
- less accidental coupling,
- cleaner internal organization.

39.2 Modules vs headers

To understand modules well, it is necessary to compare them directly with traditional headers.

39.2.1 Headers are textual inclusion

A header is a source file fragment that becomes part of each translation unit that includes it.

```cpp
// utils.hpp
#pragma once

int multiply(int a, int b);
```

```cpp
// utils.cpp
#include "utils.hpp"

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

```cpp
// main.cpp
#include "utils.hpp"
#include <iostream>

int main()
{
    std::cout << multiply(6, 7) << '\n';
}
```

This is familiar, but the compiler must parse the included declarations in every translation unit that includes the header.

39.2.2 Modules are semantic imports

A module interface unit exports declarations through the language itself.

```cpp
// utils.ixx
export module utils;

export int multiply(int a, int b)
{
    return a * b;
}
```

```cpp
// main.cpp
import utils;
#include <iostream>

int main()
{
    std::cout << multiply(6, 7) << '\n';
}
```

Here the importing translation unit does not receive pasted text from the module interface in the same way as headers do. Instead, it imports a compiled semantic interface.

39.2.3 Header guards vs module boundaries

Headers typically use include guards or `#pragma once` to avoid multiple inclusion inside one translation unit.

```
#ifndef MYLIB_HPP
#define MYLIB_HPP

int f();

#endif
```

Modules do not solve duplication through include guards. They solve it structurally by changing the language model itself.

39.2.4 Macro behavior

One of the most important differences is macro isolation.
Headers live inside the preprocessor model. This means macros can affect included declarations.

```
#define VERSION 3
#include "config.hpp"
```

That style can lead to hidden coupling and fragile behavior.
Modules are much more isolated from macro-based interference. This is one of their major conceptual advantages. While the global build still exists in a preprocessing environment, named modules are not designed around textual macro injection the way headers are.

39.2.5 Transitive exposure

Headers often create accidental transitive dependencies. For example:

```
// a.hpp
#pragma once
#include <vector>

struct A
{
    std::vector<int> data;
};
```

```
// main.cpp
#include "a.hpp"
```

Now `main.cpp` indirectly depends on whatever `a.hpp` includes. Over time, such transitive dependencies can expand significantly.
With modules, exported declarations are more intentional. The library author explicitly decides what is exported and what remains internal.

39.2.6 Interface and implementation separation

A named module often has a primary interface unit and one or more implementation units.

39.2.7 Primary interface unit

```
// geometry.ixx
export module geometry;

export struct Point
{
    int x;
    int y;
};

export int area(int width, int height);
```

39.2.8 Implementation unit

```
// geometry.cpp
module geometry;

int area(int width, int height)
{
    return width * height;
}
```

39.2.9 Consumer

```
// main.cpp
import geometry;
#include <iostream>

int main()
{
    Point p{10, 20};
    std::cout << area(p.x, p.y) << '\n';
}
```

This structure is cleaner than the classic header-plus-source arrangement in many cases because the public surface is explicitly exported.

39.2.10 Module partitions

Modules can also be split into partitions, which help organize larger modules.

```
// math.ixx
export module math;
export import :core;
export import :advanced;
```

```
// math-core.ixx
export module math:core;

export int add(int a, int b)
{
```

```
    return a + b;
}
```

```
// math-advanced.ixx
export module math:advanced;

export int square(int x)
{
    return x * x;
}
```

```
// main.cpp
import math;
#include <iostream>

int main()
{
    std::cout << add(2, 3) << '\n';
    std::cout << square(9) << '\n';
}
```

Partitions are useful for large library design, though build-system support and compiler workflows must be handled carefully.

39.2.11 Header units as a migration bridge

Header units allow certain headers to be imported. This can be useful during migration, but it should not be confused with full named-module design.
For example, some environments may support forms such as:

```
import <iostream>;
```

However, real-world support for standard library modularization and header units varies by compiler, standard library implementation, and build workflow. In serious production work, named modules should be treated as the main design target, while header units are best viewed as a transitional tool.

39.2.12 A direct comparison

- Headers are text-based. Modules are language-based.
- Headers depend heavily on preprocessing. Modules reduce that dependency.
- Headers can leak macros and transitive includes. Modules provide stronger boundaries.
- Headers are repeatedly parsed. Modules are compiled as reusable interface artifacts.
- Headers remain universal and mature. Modules are newer and require stronger toolchain discipline.

39.3 Benefits and limitations

Modules bring major benefits, but they also introduce practical limitations that every professional C++ programmer should understand.

39.3.1 Benefits

39.3.2 Faster builds in many scenarios

One of the main motivations is improved build performance. Because a module interface can be compiled once and then imported by users, repeated parsing costs can be reduced significantly in large systems.
This does not mean every small project instantly becomes faster. The biggest gains usually appear when:

- the codebase is large,
- headers are deep and widely shared,
- build dependencies are heavy,
- the build system manages module artifacts efficiently.

39.3.3 Stronger encapsulation

Modules make it easier to expose only what is intended to be public.

```
// account.ixx
export module account;

export class Account
{
public:
    Account(int id, double balance);
    int id() const;
    double balance() const;

private:
    int id_;
    double balance_;
};
```

```
// account.cpp
module account;

Account::Account(int id, double balance)
    : id_(id), balance_(balance)
{
}

int Account::id() const
{
    return id_;
}

double Account::balance() const
{
    return balance_;
}
```

This design is explicit about the exported API surface.

39.3.4 Less accidental dependency exposure

A header-based library may accidentally expose many implementation headers. Modules encourage tighter interfaces. This improves maintainability because dependent code becomes less likely to rely on details that were never intended to be public.

39.3.5 Reduced macro problems

Macros are one of the oldest global-hazard mechanisms in C and C++. Modules reduce their role in API boundaries. That does not remove macros from the language, but it reduces how much they shape library architecture.

39.3.6 Cleaner architecture

Modules encourage a more deliberate decomposition of software:

- public interface units,
- implementation units,
- optional partitions,
- controlled exported declarations.

This often improves readability of large systems.

39.3.7 Better library design habits

When writing a module, the author must think clearly about what is exported and what remains internal. This improves API design discipline.

39.3.8 A realistic larger example

39.3.9 Module interface

```
// text_utils.ixx
export module text_utils;

import <string>;
import <string_view>;
import <cctype>;

export std::string to_upper(std::string_view input)
{
    std::string result(input.begin(), input.end());

    for (char& ch : result)
        ch = static_cast<char>(
            std::toupper(static_cast<unsigned char>(ch))
        );

    return result;
}
```

39.3.10 Consumer

```
// main.cpp
import text_utils;
#include <iostream>

int main()
{
    std::cout << to_upper("Modern C++ Modules") << '\n';
}
```

The public API is compact and explicit. The consumer imports the feature by name rather than discovering it through include chains.

39.3.11 Limitations

39.3.12 Compiler support is still practical rather than perfectly uniform

Although modules are standardized in C++20, real-world support is still shaped by compiler maturity, build-system integration, standard library modularization, and vendor-specific workflows.
This means a portable modules strategy must be planned carefully. A feature may be standardized yet still require different build commands or project organization across MSVC, Clang, and GCC.

39.3.13 Build systems become more important

Headers work with very simple compilation models. Modules require the build system to understand dependencies between interface units, partitions, and implementation units.
This is one of the most important practical limitations. The language feature is not enough by itself. The surrounding tooling must know:

- which files define module interfaces,
- in what order module artifacts must be built,
- where compiled module interface files are stored,
- how import dependencies are resolved.

39.3.14 Migration is not automatic

Large existing header-based codebases do not become modular simply by replacing `#include` with `import`. Migration often requires:

- API cleanup,
- macro reduction,
- dependency auditing,
- build system redesign,

- compiler-specific testing.

In many organizations, migration happens gradually.

39.3.15 Header-based ecosystems remain dominant

A great deal of existing C++ code, third-party libraries, package managers, and tutorials is still header-based. Therefore, real projects often operate in mixed environments where both modules and headers coexist.

39.3.16 Header units are not a complete solution

Header units can help bridge old and new code, but they do not automatically deliver the full architectural clarity of named modules. They are useful, but they should not be mistaken for a complete replacement strategy.

39.3.17 Some language interactions remain subtle

Modules improve isolation, but C++ remains a rich and complex language. Interaction with templates, macros, legacy headers, build pipelines, and standard library availability may still require careful engineering judgment.

39.3.18 Not every project benefits equally

Modules are especially valuable in large systems and library-heavy builds. In very small programs, the practical benefit may be limited compared with the effort required to set up a module-aware build.

39.3.19 Professional guidance

A realistic modern approach is:

- continue understanding headers well,
- learn named modules as the long-term architectural direction,
- treat header units as a migration aid,
- design build workflows carefully,
- verify compiler-specific behavior on the actual toolchain in use.

39.3.20 Windows-oriented practical examples

The following examples show typical Windows-oriented command-line workflows. Exact file extensions and compiler behavior may vary by toolchain and project configuration, but these patterns reflect the standard structure used in current practice.

39.3.21 MSVC example

```
cl /std:c++20 /EHsc /c math.ixx
cl /std:c++20 /EHsc main.cpp math.obj
```

In real Visual Studio projects, module handling is often configured through the IDE and project system rather than handwritten command lines. Visual Studio 2022 provides direct project support for named modules.

39.3.22 Clang example on Windows

```
clang++ -std=c++20 --precompile math.ixx -o math.pcm
clang++ -std=c++20 main.cpp math.pcm -o app.exe
```

Actual workflows may differ depending on Clang version, library setup, and whether a build system such as CMake is managing the module dependency graph.

39.3.23 GCC example on Windows through MinGW or similar environments

```
g++ -std=c++20 -fmodules-ts -c math.ixx
g++ -std=c++20 -fmodules-ts main.cpp math.o -o app.exe
```

Because GCC documentation still describes incomplete or experimental parts of modules support in some areas, real GCC-based workflows should be tested carefully for the exact release in use.

39.3.24 Design advice for beginners

For a beginner learning modules inside a modern C++ book, the best order is:

- first understand traditional headers and translation units,
- then learn what problems modules solve,
- start with one simple named module,
- later study interface units, implementation units, and partitions,
- finally examine build-system integration.

Do not treat modules as magic syntax. They are a redesign of C++ program structure.

39.3.25 When to use modules

Modules are a strong fit when:

- you are starting a new modern codebase,
- you control the build system,
- compile-time scalability matters,
- clear public API boundaries matter,
- you want stronger long-term architecture.

39.3.26 When headers still remain reasonable

Headers are still reasonable when:

- you must integrate heavily with older ecosystems,
- your build environment is not yet module-ready,
- the project is small and stable,
- a third-party dependency is entirely header-based,
- portability constraints make modules difficult in the current environment.

Summary

C++ modules are one of the most important structural improvements in Modern C++. Their core motivation is to replace the fragile and expensive textual inclusion model with a semantic import model.
Compared with headers, modules provide:

- stronger encapsulation,
- reduced macro interference,
- clearer dependency boundaries,
- potential build-time improvements,
- a more deliberate interface-oriented architecture.

At the same time, modules are not a magic replacement for all existing code. They depend heavily on practical compiler support, build-system integration, and careful migration strategy.
For modern professional C++, the correct mental model is not “headers are obsolete immediately,” but rather “modules are the long-term architectural direction, and serious C++ programmers should understand both models deeply.”
Headers explain where C++ came from. Modules explain where modern C++ is going.

Coroutines

40.1 Fundamentals

Coroutines were introduced in C++20 as a language-level facility for writing asynchronous and lazy code in a sequential style. Unlike threads, coroutines do not represent concurrent execution by default. Instead, they represent **suspendable functions** that can pause and resume execution at well-defined points.
A coroutine is a function that uses one of the coroutine keywords:

- `co_await`
- `co_yield`
- `co_return`

When a function contains any of these, the compiler transforms it into a state machine.

40.1.1 Key idea

A coroutine:

- can suspend execution,
- can resume later,
- preserves its local state between suspensions,
- is controlled by a **promise type** and a **coroutine handle**.

40.1.2 How coroutines differ from functions

A normal function:

- executes from start to end,
- returns once,
- does not preserve execution state after returning.

A coroutine:

- may suspend multiple times,

- resumes where it left off,
- maintains its internal state in a compiler-generated frame.

40.1.3 Coroutine transformation model

When a coroutine is compiled, the compiler generates:

- a coroutine frame (heap or optimized storage),
- a promise object,
- suspension points,
- a state machine controlling execution flow.

The programmer does not manually write this state machine. The compiler generates it automatically.

40.1.4 Basic coroutine structure

A coroutine requires a return type that defines a nested `promise_type`.

```
#include <coroutine>

struct Task
{
    struct promise_type
    {
        Task get_return_object()
        {
            return {};
        }

        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }

        void return_void() {}
        void unhandled_exception() {}
    };
};

Task simple_coroutine()
{
    co_return;
}
```

This is the minimal structure required for a coroutine.

40.1.5 Suspend points

Suspension is controlled through special awaitable objects. Two standard helpers are:

- `std::suspend_never`
- `std::suspend_always`

```
#include <coroutine>
#include <iostream>

struct SimpleTask
{
    struct promise_type
    {
        SimpleTask get_return_object() { return {}; }

        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }

        void return_void() {}
        void unhandled_exception() {}
    };
};

SimpleTask demo()
{
    std::cout << "Before suspend\n";
    co_await std::suspend_always{};
    std::cout << "After resume\n";
}
```

The coroutine pauses at `co_await` and resumes later.

40.1.6 Coroutine handle

The runtime representation of a coroutine is accessed through `std::coroutine_handle`.

```
#include <coroutine>

std::coroutine_handle<> handle;
```

The handle allows:

- resuming execution,
- destroying the coroutine,
- accessing the promise.

40.1.7 Memory model

A coroutine allocates a frame that stores:

- local variables,
- parameters,

- promise object,
- suspension state.

The allocation strategy may vary depending on optimization and compiler implementation. In many cases, heap allocation is used, but compilers can optimize to stack or elide allocation.

40.2 `co_await`, `co_yield`, `co_return`

These three keywords define the behavior of coroutines.

40.2.1 `co_await`

`co_await` is used to suspend a coroutine until a condition is satisfied. It works with **awaitable** objects.
An awaitable object must define:

- `await_ready()`
- `await_suspend()`
- `await_resume()`

40.2.2 Custom awaitable example

```
#include <coroutine>
#include <iostream>

struct Awaiter
{
    bool await_ready() const noexcept
    {
        return false;
    }

    void await_suspend(std::coroutine_handle<> h)
    {
        std::cout << "Suspending coroutine\n";
        h.resume();
    }

    int await_resume() const noexcept
    {
        return 42;
    }
};

struct Task
{
    struct promise_type
    {
        Task get_return_object() { return {}; }
```

```
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }

        void return_void() {}
        void unhandled_exception() {}
    };
};

Task example()
{
    int value = co_await Awaiter{};
    std::cout << "Resumed with value: " << value << "\n";
}
```

40.2.3 Behavior of `co_await`

- checks `await_ready()`,
- suspends if necessary,
- resumes when ready,
- returns result via `await_resume()`.

40.2.4 `co_yield`

`co_yield` is used for producing values in generator-style coroutines.

```
#include <coroutine>
#include <iostream>

template<typename T>
struct Generator
{
    struct promise_type
    {
        T current_value;

        Generator get_return_object()
        {
            return Generator{
                std::coroutine_handle<promise_type>::from_promise(*this)
            };
        }

        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }

        std::suspend_always yield_value(T value)
        {
            current_value = value;
```

```
            return {};
        }

        void return_void() {}
        void unhandled_exception() {}
    };

    std::coroutine_handle<promise_type> handle;

    Generator(std::coroutine_handle<promise_type> h) : handle(h) {}

    ~Generator()
    {
        if (handle)
            handle.destroy();
    }

    bool next()
    {
        handle.resume();
        return !handle.done();
    }

    T value() const
    {
        return handle.promise().current_value;
    }
};

Generator<int> counter()
{
    for (int i = 0; i < 5; ++i)
        co_yield i;
}
```

40.2.5 Using the generator

```
#include <iostream>

int main()
{
    auto gen = counter();

    while (gen.next())
        std::cout << gen.value() << "\n";
}
```

40.2.6 `co_return`

`co_return` ends the coroutine and returns a value (or no value).

```
#include <coroutine>
```

```
struct Task
{
    struct promise_type
    {
        Task get_return_object() { return {}; }

        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }

        void return_value(int) {}
        void unhandled_exception() {}
    };
};

Task compute()
{
    co_return 10;
}
```

40.2.7 Differences summary

- `co_await`: suspend until ready.
- `co_yield`: produce a sequence of values.
- `co_return`: finish coroutine execution.

40.3 Use cases

Coroutines are not a general replacement for threads. They are best suited for structured asynchronous workflows, lazy evaluation, and stateful iteration.

40.3.1 Asynchronous programming

Coroutines simplify asynchronous code by avoiding deeply nested callbacks.
Traditional style:

```
// pseudo-code
async_read(socket, buffer, handler1);
```

Coroutine style:

```
// conceptual
auto data = co_await async_read(socket, buffer);
```

This makes code easier to read and maintain.

40.3.2 Networking and I/O

Libraries such as asynchronous I/O frameworks can integrate with coroutines.

- non-blocking sockets,
- asynchronous file operations,
- event-driven systems.

Coroutines allow writing such systems in a linear style while preserving asynchronous behavior.

40.3.3 Lazy generators

Generators are one of the most natural uses of `co_yield`.

```
Generator<int> fibonacci()
{
    int a = 0, b = 1;

    while (true)
    {
        co_yield a;
        int next = a + b;
        a = b;
        b = next;
    }
}
```

This produces an infinite sequence lazily.

40.3.4 Pipelines and streaming

Coroutines can be used to process streams of data:

- parsing streams,
- transforming sequences,
- incremental processing of large datasets.

40.3.5 State machines

Coroutines naturally model state machines without explicit state variables.

```
Task process()
{
    co_await step1();
    co_await step2();
    co_await step3();
}
```

This replaces complex manual state handling with linear logic.

40.3.6 Game loops and simulations

In game engines or simulations, coroutines can represent entities that pause and resume over time.

40.3.7 Embedded and systems programming

Coroutines can be used in:

- event loops,
- cooperative multitasking,
- resource-constrained asynchronous systems.

40.3.8 Performance considerations

- Coroutines avoid thread overhead.
- They can reduce context-switch cost.
- Memory allocation for coroutine frames must be considered.
- Excessive coroutine usage may increase complexity.

40.3.9 Design guidelines

- Use coroutines for asynchronous flows, not for simple synchronous logic.
- Keep coroutine lifetimes clear and well-managed.
- Avoid hidden control flow that reduces readability.
- Use RAII carefully with coroutine suspension.
- Understand ownership of coroutine handles.

40.3.10 Common pitfalls

- forgetting to destroy coroutine handles,
- misunderstanding suspension behavior,
- mixing blocking and coroutine-based logic incorrectly,
- overusing coroutines where simpler code is sufficient.

Summary

Coroutines introduce a powerful abstraction for writing asynchronous and lazy code in Modern C++.

- They allow suspension and resumption of execution.
- They are built around compiler-generated state machines.
- `co_await`, `co_yield`, and `co_return` define their behavior.
- They enable cleaner asynchronous and generator-based designs.

Used correctly, coroutines significantly improve readability and structure of complex asynchronous systems while maintaining high performance and control over execution.

Key Modern Features Overview

41.1 `std::span`

`std::span` is one of the most important practical features added in Modern C++. It provides a lightweight, non-owning view over a contiguous sequence of objects. A span does not allocate memory, does not own memory, and does not copy elements. Instead, it refers to existing storage and exposes that storage through a safe and expressive interface.
This makes `std::span` ideal for functions that need to accept arrays, `std::array`, `std::vector`, C-style arrays, or any other contiguous memory block without forcing a particular container type.

41.1.1 Why `std::span` matters

Before `std::span`, APIs often used one of the following patterns:

- raw pointer + size,
- reference to `std::vector`,
- reference to `std::array`,
- iterator pair.

Each of those has weaknesses:

- raw pointer + size is error-prone,
- a specific container type reduces generality,
- iterator pairs do not directly express contiguous storage,
- overload sets become large when many container types must be supported.

`std::span` solves this elegantly.

41.1.2 Basic example

```
#include <iostream>
#include <span>
#include <vector>
#include <array>
```

```
void print_values(std::span<const int> values)
{
    for (int v : values)
        std::cout << v << ' ';
    std::cout << '\n';
}

int main()
{
    int raw[] = {1, 2, 3, 4};
    std::array<int, 3> arr = {10, 20, 30};
    std::vector<int> vec = {100, 200, 300, 400};

    print_values(raw);
    print_values(arr);
    print_values(vec);
}
```

The function accepts all three without overload duplication.

41.1.3 Static extent and dynamic extent

A span can have either:

- a compile-time known extent,
- or a runtime extent.

```
#include <span>

void fixed_buffer(std::span<int, 4> s)
{
}

void variable_buffer(std::span<int> s)
{
}
```

A fixed-extent span can be useful when the interface requires an exact size.

41.1.4 Subspans

```
#include <iostream>
#include <span>

int main()
{
    int data[] = {10, 20, 30, 40, 50};

    std::span<int> s(data);

    auto first_three = s.first<3>();
```

```
    auto last_two = s.last(2);
    auto middle = s.subspan(1, 3);

    for (int x : first_three) std::cout << x << ' ';
    std::cout << '\n';

    for (int x : last_two) std::cout << x << ' ';
    std::cout << '\n';

    for (int x : middle) std::cout << x << ' ';
    std::cout << '\n';
}
```

41.1.5 Using `as_bytes` and `as_writable_bytes`

The header also provides helpers to view the object representation as bytes.

```
#include <iostream>
#include <span>
#include <cstddef>

int main()
{
    int values[] = {1, 2, 3};
    std::span<int> s(values);

    auto bytes = std::as_bytes(s);

    std::cout << "Byte count: " << bytes.size() << '\n';
}
```

This is useful for serialization, hashing, packet inspection, and low-level diagnostics. It should still be used with a clear understanding of object representation and portability limits.

41.1.6 Best practices

- Use `std::span` for read-only input with `std::span<const T>`.
- Use `std::span<T>` when mutation is intended.
- Do not return a span that refers to destroyed storage.
- Remember that span does not own data.
- Prefer span over raw pointer + length for contiguous buffers.

41.2 `std::source_location`

`std::source_location` is a C++20 facility designed for diagnostics, tracing, logging, assertions, and debugging support. It allows a function to capture information about the call site without requiring preprocessor macros such as `__FILE__` and `__LINE__` in every user call.

41.2.1 Why it matters

Before `std::source_location`, logging helpers often used macros:

```
#define LOG(msg) log_impl(msg, __FILE__, __LINE__)
```

This works, but macros have well-known disadvantages:

- weak type safety,
- harder debugging,
- less clean APIs,
- reduced readability.

`std::source_location` allows the same goal with ordinary function interfaces.

41.2.2 Basic example

```
#include <iostream>
#include <source_location>
#include <string_view>

void log_message(
    std::string_view msg,
    const std::source_location& loc = std::source_location::current())
{
    std::cout
        << loc.file_name() << ':'
        << loc.line() << ' '
        << loc.function_name() << " -> "
        << msg << '\n';
}

int main()
{
    log_message("Application started");
}
```

The call site is captured automatically.

41.2.3 Practical logging helper

```
#include <iostream>
#include <source_location>
#include <string_view>

enum class LogLevel
{
    info,
    warning,
```

```
    error
};

void log(
    LogLevel level,
    std::string_view msg,
    const std::source_location& loc = std::source_location::current())
{
    const char* level_name = "";

    switch (level)
    {
        case LogLevel::info: level_name = "INFO"; break;
        case LogLevel::warning: level_name = "WARNING"; break;
        case LogLevel::error: level_name = "ERROR"; break;
    }

    std::cout
        << '[' << level_name << "] "
        << loc.file_name() << ':'
        << loc.line() << ' '
        << msg << '\n';
}

int main()
{
    log(LogLevel::info, "Server initialization");
    log(LogLevel::warning, "Configuration file not found, using defaults");
    log(LogLevel::error, "Socket creation failed");
}
```

41.2.4 Design guidance

- Use `std::source_location` in diagnostics APIs.
- Pass it as a defaulted final parameter.
- Prefer it over macros when ordinary functions are sufficient.
- Do not rely on exact formatting of file names across toolchains.

41.3 `std::format`

`std::format` is a modern formatting facility that replaces many use cases of `printf`, string concatenation, and stream-based formatting. It is type-safe, expressive, and much easier to read in many scenarios.

41.3.1 Why `std::format` matters

Traditional formatting methods in C++ often had one of these drawbacks:

- `printf` is not type-safe,

- `std::ostringstream` can be verbose,
- concatenation can become hard to read,
- iostream manipulators can leave persistent formatting state behind.

`std::format` gives modern C++ a direct formatting API based on replacement fields.

41.3.2 Basic example

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

int main()
{
    std::string name = "Ayman";
    int version = 23;

    std::string text = std::format("Hello, {}. C++ version marker: {}", name, version);
    std::cout << text << '\n';
}
```

41.3.3 Numeric formatting

```
#include <format>
#include <iostream>

int main()
{
    int value = 255;

    std::cout << std::format("decimal: {}\n", value);
    std::cout << std::format("hex: {:x}\n", value);
    std::cout << std::format("HEX: {:X}\n", value);
    std::cout << std::format("binary: {:b}\n", value);
    std::cout << std::format("padded: {:08}\n", value);
}
```

41.3.4 Floating-point formatting

```
#include <format>
#include <iostream>

int main()
{
    double pi = 3.141592653589793;

    std::cout << std::format("default: {}\n", pi);
    std::cout << std::format("fixed 2: {:.2f}\n", pi);
    std::cout << std::format("scientific: {:.4e}\n", pi);
}
```

41.3.5 Alignment and width

```
#include <format>
#include <iostream>

int main()
{
    std::cout << std::format("|{:>10}|\n", "right");
    std::cout << std::format("|{:<10}|\n", "left");
    std::cout << std::format("|{:^10}|\n", "center");
}
```

41.3.6 Formatting ranges

With current C++23 library evolution, range formatting support and improvements are part of the modern direction of `std::format`. On toolchains that support range formatting, containers can be formatted directly.

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

int main()
{
    std::vector<int> values = {1, 2, 3, 4};

    // Requires library support for range formatting
    std::cout << std::format("{}\n", values);
}
```

When portability is critical, confirm exact support on the target compiler and standard library.

41.3.7 Custom formatter example

A type can define a formatter specialization.

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

struct Point
{
    int x;
    int y;
};

template<>
struct std::formatter<Point> : std::formatter<std::string>
{
    auto format(const Point& p, std::format_context& ctx) const
    {
        return std::formatter<std::string>::format(
            std::format("({}, {})", p.x, p.y), ctx);
```

```
    }
};

int main()
{
    Point p{10, 20};
    std::cout << std::format("Point = {}\n", p);
}
```

41.3.8 Guidelines

- Prefer `std::format` for string creation and structured textual output.
- Use explicit format specifiers for clarity.
- Validate library support on the exact Windows compiler and standard library you deploy.
- Prefer custom formatters for user-defined types instead of ad hoc conversion helpers.

41.4 `std::print`

`std::print` is a C++23 formatted output facility built on the formatting model of `std::format`. It is intended for direct printing to standard output or streams in a simpler and more expressive way than repeated use of `std::cout` insertion chains.

41.4.1 Why `std::print` matters

`std::print` removes a common inconvenience in C++:

```
std::cout << "x = " << x << ", y = " << y << '\n';
```

Instead:

```
std::print("x = {}, y = {}\n", x, y);
```

This is shorter, more regular, and consistent with `std::format`.

41.4.2 Basic example

```
#include <print>

int main()
{
    int id = 42;
    double value = 18.75;

    std::print("id = {}, value = {:.2f}\n", id, value);
}
```

41.4.3 Printing to `stderr`

```
#include <print>
#include <cstdio>

int main()
{
    std::print(stderr, "Error: cannot open file '{}'\n", "config.txt");
}
```

41.4.4 `std::println`

Many modern implementations also provide `std::println`, which appends a newline automatically.

```
#include <print>

int main()
{
    std::println("Hello from {}", "Modern C++");
}
```

41.4.5 Practical notes

- `std::print` depends on library support, not just parser support.
- On Windows, compile with a mode that enables C++23 or latest-preview support as required by the active toolchain.
- When deployment portability matters, verify the exact standard library implementation in use.

41.4.6 Useful pattern

```
#include <print>
#include <string_view>

void report_progress(std::string_view task, int done, int total)
{
    std::print("[{}] {}/{}\n", task, done, total);
}

int main()
{
    report_progress("Parsing", 10, 100);
    report_progress("Compiling", 50, 100);
    report_progress("Linking", 100, 100);
}
```

41.5 `std::expected`

`std::expected` is a C++23 vocabulary type for functions that may either produce a value or report an error, without using exceptions as the primary transport mechanism.
Conceptually, `std::expected<T, E>` contains either:

- a successful value of type `T`,
- or an unexpected error object of type `E`.

41.5.1 Why `std::expected` matters

Many programs need explicit failure handling but do not want to use:

- raw error codes everywhere,
- out parameters,
- exception-heavy interfaces,
- loosely structured sentinel values.

`std::expected` gives a direct and expressive model.

41.5.2 Basic example

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

std::expected<int, std::string> parse_positive(int x)
{
    if (x > 0)
        return x;

    return std::unexpected("value must be positive");
}

int main()
{
    auto result = parse_positive(-5);

    if (result)
        std::cout << "Value = " << *result << '\n';
    else
        std::cout << "Error = " << result.error() << '\n';
}
```

41.5.3 File-open style example

```
#include <expected>
#include <fstream>
#include <string>

std::expected<std::ifstream, std::string> open_input(const std::string& path)
{
    std::ifstream in(path);
```

```
    if (!in)
        return std::unexpected("failed to open file");

    return std::move(in);
}
```

41.5.4 Chained logic

Monadic operations were added for `std::expected` in later C++23 library evolution and are part of the modern intended usage style.

```
#include <expected>
#include <string>

std::expected<int, std::string> parse_value()
{
    return 10;
}

std::expected<int, std::string> double_value(int x)
{
    return x * 2;
}

int main()
{
    auto result =
        parse_value()
        .and_then(double_value)
        .transform([](int x) { return x + 1; });

    if (result)
    {
        // result contains 21
    }
}
```

41.5.5 When to use `expected`

- parsing,
- file and configuration loading,
- protocol decoding,
- validation-heavy APIs,
- systems code where explicit success or error flow is preferred.

41.5.6 When not to force it

- very simple code where a boolean is enough,

- APIs already designed around exceptions,
- performance-sensitive paths where a different error strategy is required by architecture.

41.6 `std::mdspan`

`std::mdspan` is a C++23 multidimensional non-owning array view. It extends the philosophy of `std::span` to multidimensional indexing. It does not own storage; instead, it maps a multidimensional index space onto underlying data. This is one of the most important modern features for scientific computing, numerical code, image processing, matrix operations, simulation systems, and high-performance memory-layout-aware programming.

41.6.1 Why `std::mdspan` matters

Before `std::mdspan`, multidimensional storage often used:

- nested containers,
- manual index math,
- custom matrix wrappers,
- library-specific tensor abstractions.

`std::mdspan` provides a standard library abstraction for multidimensional views.

41.6.2 Basic example

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

int main()
{
    std::vector<int> storage(6);

    for (int i = 0; i < 6; ++i)
        storage[i] = i + 1;

    std::mdspan<int, std::extents<std::size_t, 2, 3>> m(storage.data());

    for (std::size_t r = 0; r < 2; ++r)
    {
        for (std::size_t c = 0; c < 3; ++c)
            std::cout << m[r, c] << ' ';
        std::cout << '\n';
    }
}
```

This example treats linear storage as a 2×3 matrix.

41.6.3 Dynamic extents

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

int main()
{
    std::size_t rows = 3;
    std::size_t cols = 4;

    std::vector<double> data(rows * cols, 1.5);

    std::mdspan<double,
        std::dextents<std::size_t, 2>> matrix(data.data(), rows, cols);

    std::cout << matrix[2, 3] << '\n';
}
```

41.6.4 Layouts

`std::mdspan` supports layout policies such as:

- `layout_right`,
- `layout_left`,
- `layout_stride`.

This is essential for interoperability with row-major and column-major memory layouts or with custom strided layouts.

```
#include <mdspan>
#include <vector>

int main()
{
    std::vector<int> data(12);

    std::mdspan<int,
        std::extents<std::size_t, 3, 4>,
        std::layout_right> a(data.data());

    std::mdspan<int,
        std::extents<std::size_t, 3, 4>,
        std::layout_left> b(data.data());
}
```

41.6.5 Practical use case

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

```cpp
void fill_identity(
    std::mdspan<double, std::dextents<std::size_t, 2>> m)
{
    for (std::size_t r = 0; r < m.extent(0); ++r)
    {
        for (std::size_t c = 0; c < m.extent(1); ++c)
            m[r, c] = (r == c) ? 1.0 : 0.0;
    }
}

int main()
{
    std::vector<double> storage(9);
    std::mdspan<double, std::dextents<std::size_t, 2>> mat(storage.data(), 3, 3);

    fill_identity(mat);

    for (std::size_t r = 0; r < 3; ++r)
    {
        for (std::size_t c = 0; c < 3; ++c)
            std::cout << mat[r, c] << ' ';
        std::cout << '\n';
    }
}
```

41.6.6 Why it is important for serious C++

- standard multidimensional abstraction,
- non-owning and lightweight,
- layout-aware,
- suitable for HPC and data-oriented design,
- excellent fit for reusable numerical kernels.

41.7 Overview of C++26 direction (reflection, contracts)

C++26 direction is best understood carefully. Some features are being actively standardized and tracked, but toolchain implementation is still evolving. Two of the most discussed directions are reflection and contracts.

41.7.1 Reflection

Reflection aims to let a program reason about its own structure in a standardized way. In broad terms, this means language-supported access to information about declarations, types, names, members, and other program entities.
If standardized and broadly implemented, reflection could transform:

- serialization,

- automatic bindings,
- code generation helpers,
- diagnostics,
- testing frameworks,
- metaprogramming ergonomics.

Today, C++ often simulates such capabilities through:

- macros,
- template tricks,
- external code generators,
- hand-written registration systems.

Standard reflection could greatly simplify those patterns.

41.7.2 Why reflection matters

Imagine writing a serializer without manually listing every field:

```
struct User
{
    int id;
    std::string name;
    double score;
};
```

In current mainstream C++, one often writes custom glue code. Reflection aims toward a future where tools and libraries can inspect such structure more directly and safely.

41.7.3 Contracts

Contracts aim to express program expectations explicitly in the language itself. At a high level, they are about stating conditions such as:

- what must be true before a function is called,
- what a function guarantees after completion,
- what invariants should always hold.

These ideas are already central to good software engineering, but in current C++ they are usually expressed through:

- comments,
- assertions,

- documentation,
- manual runtime checks,
- guidelines and code reviews.

Contracts aim to make these expectations more formal and more visible.

41.7.4 Why contracts matter

A function such as the following has a clear semantic rule:

```
int divide(int a, int b)
{
    return a / b;
}
```

The caller must not pass zero as the divisor. In ordinary C++, that rule exists mostly in documentation or defensive checks. Contracts aim to let the language express such intent more directly.

41.7.5 Current practical reality

For current production code, reflection and contracts should be treated as part of the modern direction of the language rather than universally available everyday facilities. Serious C++ developers should follow them closely, but should still design production code using currently implemented tools:

- concepts,
- `constexpr`,
- `consteval`,
- `std::span`,
- `std::expected`,
- assertions,
- static analysis,
- disciplined API design.

41.7.6 What to teach beginners now

For a beginner, the best approach is:

- master currently standardized and available features first,
- understand why reflection and contracts are desirable,
- watch compiler support evolve,
- write code today in a style that will benefit from those future capabilities.

41.8 Windows compilation notes

Because these features span C++20, C++23, and ongoing post-C++23 implementation work, Windows users should compile with the correct language mode and verify their standard library support.

41.8.1 MSVC

```
cl /std:c++20 /EHsc main.cpp
```

For newer library and preview work:

```
cl /std:c++latest /EHsc main.cpp
```

Some environments also use:

```
cl /std:c++23preview /EHsc main.cpp
```

depending on the installed Visual Studio toolset and project configuration.

41.8.2 Clang on Windows

```
clang++ -std=c++20 -Wall -Wextra -pedantic main.cpp -o app.exe
```

For newer post-C++23 experimentation:

```
clang++ -std=c++2c -Wall -Wextra -pedantic main.cpp -o app.exe
```

41.8.3 GCC on Windows

```
g++ -std=c++23 -Wall -Wextra -pedantic main.cpp -o app.exe
```

For newer work where the installed toolchain supports it:

```
g++ -std=c++2c -Wall -Wextra -pedantic main.cpp -o app.exe
```

41.8.4 Practical reminder

Availability of `std::format`, `std::print`, `std::expected`, and `std::mdspan` depends on the actual compiler and standard library implementation in the build environment. In professional work, always verify the exact toolchain rather than assuming that parser support alone guarantees full library availability.

Summary

This chapter collected several of the most important modern facilities in todays C++ landscape.
`std::span` improves buffer APIs by providing a safe non-owning contiguous view. `std::source_location` modernizes diagnostics and logging interfaces. `std::format` brings safe and expressive formatting. `std::print` extends that model to direct output. `std::expected` gives explicit value-or-error handling. `std::mdspan` brings a standard multidimensional non-owning view for serious numerical and systems work. Reflection and contracts represent major parts of the C++26 direction, even though implementation maturity is still evolving.
Together, these features show the real trajectory of Modern C++: safer interfaces, clearer intent, stronger abstraction, less boilerplate, and better tools for building high-performance software without giving up control.

Part XII

Building Real Projects

Project: Calculator

42.1 Project overview

In this chapter, we build a real project: a small but well-structured calculator written in Modern C++. Although the project is intentionally compact, it introduces several professional programming ideas that appear again and again in real software systems:

- separating input, parsing, and evaluation,
- validating user input carefully,
- reporting errors clearly,
- using standard library facilities instead of unsafe manual techniques,
- designing code so that it can grow into a larger project later.

A calculator is an excellent first real project because it looks simple from the outside, but it forces the programmer to think about program structure. Even a basic expression such as:

```
2 + 3 * 4
```

immediately raises important questions:

- How do we read the whole line correctly?
- How do we break it into tokens?
- How do we handle spaces?
- How do we respect operator precedence?
- How do we detect invalid expressions?
- How do we keep the program clean and extensible?

This chapter answers all of those questions step by step.

42.2 Project goals

The calculator in this chapter will support:

- integer and floating-point numbers,
- the operators `+`, `-`, `*`, and `/`,
- parentheses,
- spaces anywhere in the expression,
- repeated interactive input in a loop,
- graceful error handling.

Examples of valid input:

```
1 + 2
10 - 3 * 2
(4 + 5) * 6
18 / 3 + 7
2.5 * 4
```

Examples of invalid input:

```
1 +
* 3
(2 + 5
10 / 0
abc
```

The goal is not merely to compute answers. The goal is to design the calculator in a way that teaches real software engineering.

42.3 Why this project matters

Many beginners write calculators in a single long function. That can work for very small exercises, but it usually leads to several problems:

- logic becomes mixed together,
- parsing becomes fragile,
- error reporting becomes inconsistent,
- extending the program later becomes difficult.

A better design separates the project into clear stages:

1. read a line of text,
2. convert text into tokens,

3. parse tokens into an expression structure,
4. evaluate that structure,
5. print the result or an error message.

This layered design is much closer to how professional tools are written.

42.4 High-level design

Our calculator will use a simple recursive-descent parser. This is one of the clearest parsing techniques for beginners and is also widely used in real compilers, interpreters, and domain-specific tools.
We define the expression grammar in a structured way:

```
expression = term { ('+' | '-') term }
term       = factor { ('*' | '/') factor }
factor     = number | '(' expression ')' | unary minus
```

This grammar naturally gives multiplication and division higher precedence than addition and subtraction.
For example:

```
2 + 3 * 4
```

is interpreted as:

```
2 + (3 * 4)
```

not:

```
(2 + 3) * 4
```

42.5 Program architecture

We will organize the calculator using the following components:

- `TokenType`: identifies what kind of token we have,
- `Token`: stores one token,
- `Tokenizer`: converts text into tokens,
- `Parser`: parses and evaluates expressions,
- `main()`: runs the interactive calculator loop.

This design is intentionally simple, but already demonstrates the idea of dividing responsibilities.

42.6 Token representation

The first step is tokenization. A token is a small unit such as a number, operator, or parenthesis.
We begin with an enumeration for token types.

```
enum class TokenType
{
    Number,
    Plus,
    Minus,
    Star,
    Slash,
    LeftParen,
    RightParen,
    End
};
```

Next, we define a token structure.

```
#include <string>

struct Token
{
    TokenType type{};
    double value{};
    std::string text{};
};
```

The fields mean:

- `type`: the category of token,
- `value`: numeric value when the token is a number,
- `text`: original text for diagnostics.

42.7 The tokenizer

The tokenizer reads a string and produces a sequence of tokens.
A professional advantage of tokenization is that it separates character-by-character processing from parsing logic. The parser can then work with meaningful units instead of raw characters.

42.7.1 Tokenizer implementation

```
#include <charconv>
#include <cctype>
#include <stdexcept>
#include <string>
#include <string_view>
#include <vector>
```

```
class Tokenizer
{
public:
    explicit Tokenizer(std::string_view input)
        : input_(input), pos_(0)
    {
    }

    std::vector<Token> tokenize()
    {
        std::vector<Token> tokens;

        while (true)
        {
            skip_spaces();

            if (pos_ >= input_.size())
            {
                tokens.push_back(Token{TokenType::End, 0.0, ""});
                break;
            }

            char ch = input_[pos_];

            switch (ch)
            {
                case '+':
                    tokens.push_back(Token{TokenType::Plus, 0.0, "+"});
                    ++pos_;
                    break;

                case '-':
                    tokens.push_back(Token{TokenType::Minus, 0.0, "-"});
                    ++pos_;
                    break;

                case '*':
                    tokens.push_back(Token{TokenType::Star, 0.0, "*"});
                    ++pos_;
                    break;

                case '/':
                    tokens.push_back(Token{TokenType::Slash, 0.0, "/"});
                    ++pos_;
                    break;

                case '(':
                    tokens.push_back(Token{TokenType::LeftParen, 0.0, "("});
                    ++pos_;
                    break;
```

```
                case ')':
                    tokens.push_back(Token{TokenType::RightParen, 0.0, ")"});
                    ++pos_;
                    break;

                default:
                    if (std::isdigit(static_cast<unsigned char>(ch)) || ch == '.')
                    {
                        tokens.push_back(read_number());
                    }
                    else
                    {
                        throw std::runtime_error(
                            "Unexpected character: " + std::string(1, ch));
                    }
                    break;
            }
        }

        return tokens;
    }

private:
    Token read_number()
    {
        std::size_t start = pos_;
        bool dot_seen = false;

        while (pos_ < input_.size())
        {
            char ch = input_[pos_];

            if (std::isdigit(static_cast<unsigned char>(ch)))
            {
                ++pos_;
            }
            else if (ch == '.')
            {
                if (dot_seen)
                    break;

                dot_seen = true;
                ++pos_;
            }
            else
            {
                break;
            }
        }

        std::string text(input_.substr(start, pos_ - start));
```

```
        double value{};
        auto result = std::from_chars(
            text.data(),
            text.data() + text.size(),
            value
        );

        if (result.ec != std::errc{})
        {
            throw std::runtime_error("Invalid number: " + text);
        }

        return Token{TokenType::Number, value, text};
    }

    void skip_spaces()
    {
        while (pos_ < input_.size() &&
               std::isspace(static_cast<unsigned char>(input_[pos_])))
        {
            ++pos_;
        }
    }

private:
    std::string_view input_;
    std::size_t pos_;
};
```

42.7.2 Why this tokenizer is good

This tokenizer is good for several reasons:

- it skips spaces cleanly,
- it reports unexpected characters,
- it uses `std::from_chars` for numeric conversion,
- it separates lexical processing from grammar logic.

42.7.3 Using `std::from_chars`

Modern C++ provides `std::from_chars` as a low-level numeric conversion utility. For parser-like tasks, this is often preferable to older conversion techniques because it is direct and explicit.
For this calculator, it gives us a clean way to convert numeric substrings such as:

```
123
45.67
0.5
```

into numeric values.

42.8 Parser fundamentals

After tokenization, the parser reads tokens according to grammar rules.
A recursive-descent parser uses one function per grammar level. This leads to code that maps directly to the grammar.
Our parser functions will be:

- `parse_expression()`
- `parse_term()`
- `parse_factor()`

42.8.1 Parser implementation

```
#include <stdexcept>
#include <vector>

class Parser
{
public:
    explicit Parser(std::vector<Token> tokens)
        : tokens_(std::move(tokens)), pos_(0)
    {
    }

    double parse()
    {
        double result = parse_expression();

        if (current().type != TokenType::End)
        {
            throw std::runtime_error(
                "Unexpected token after complete expression: " + current().text);
        }

        return result;
    }

private:
    double parse_expression()
    {
        double left = parse_term();

        while (current().type == TokenType::Plus ||
               current().type == TokenType::Minus)
        {
            TokenType op = current().type;
            advance();

            double right = parse_term();
```

```
        if (op == TokenType::Plus)
            left += right;
        else
            left -= right;
    }

    return left;
}

double parse_term()
{
    double left = parse_factor();

    while (current().type == TokenType::Star ||
           current().type == TokenType::Slash)
    {
        TokenType op = current().type;
        advance();

        double right = parse_factor();

        if (op == TokenType::Star)
        {
            left *= right;
        }
        else
        {
            if (right == 0.0)
                throw std::runtime_error("Division by zero");

            left /= right;
        }
    }

    return left;
}

double parse_factor()
{
    if (current().type == TokenType::Minus)
    {
        advance();
        return -parse_factor();
    }

    if (current().type == TokenType::Number)
    {
        double value = current().value;
        advance();
        return value;
    }
```

```
        if (current().type == TokenType::LeftParen)
        {
            advance();
            double value = parse_expression();

            if (current().type != TokenType::RightParen)
                throw std::runtime_error("Missing closing parenthesis");

            advance();
            return value;
        }

        throw std::runtime_error("Expected number or '('");
    }

    const Token& current() const
    {
        return tokens_[pos_];
    }

    void advance()
    {
        if (pos_ < tokens_.size())
            ++pos_;
    }

private:
    std::vector<Token> tokens_;
    std::size_t pos_;
};
```

42.9 How the parser works

Let us trace:

```
2 + 3 * 4
```

The tokenizer produces:

```
Number(2), Plus, Number(3), Star, Number(4), End
```

Then parsing proceeds as follows:

1. `parse_expression()` reads the first term.
2. That term begins with factor `2`.
3. The parser sees `+`, so it must parse another term.
4. That second term begins with factor `3`.
5. It then sees `*`, so it multiplies by factor `4`.

6. The second term becomes `12`.
7. The whole expression becomes `2 + 12 = 14`.

This is exactly why grammar structure matters. The parser is not using a special precedence table here. The precedence naturally comes from the separation of expression, term, and factor functions.

42.10 Full program

Now we combine the tokenizer, parser, and interactive loop into one complete program.

```
#include <charconv>
#include <cctype>
#include <errc>
#include <iostream>
#include <stdexcept>
#include <string>
#include <string_view>
#include <vector>

enum class TokenType
{
    Number,
    Plus,
    Minus,
    Star,
    Slash,
    LeftParen,
    RightParen,
    End
};

struct Token
{
    TokenType type{};
    double value{};
    std::string text{};
};

class Tokenizer
{
public:
    explicit Tokenizer(std::string_view input)
        : input_(input), pos_(0)
    {
    }

    std::vector<Token> tokenize()
    {
        std::vector<Token> tokens;
```

```
        while (true)
        {
            skip_spaces();

            if (pos_ >= input_.size())
            {
                tokens.push_back(Token{TokenType::End, 0.0, ""});
                break;
            }

            char ch = input_[pos_];

            switch (ch)
            {
                case '+':
                    tokens.push_back(Token{TokenType::Plus, 0.0, "+"});
                    ++pos_;
                    break;

                case '-':
                    tokens.push_back(Token{TokenType::Minus, 0.0, "-"});
                    ++pos_;
                    break;

                case '*':
                    tokens.push_back(Token{TokenType::Star, 0.0, "*"});
                    ++pos_;
                    break;

                case '/':
                    tokens.push_back(Token{TokenType::Slash, 0.0, "/"});
                    ++pos_;
                    break;

                case '(':
                    tokens.push_back(Token{TokenType::LeftParen, 0.0, "("});
                    ++pos_;
                    break;

                case ')':
                    tokens.push_back(Token{TokenType::RightParen, 0.0, ")"});
                    ++pos_;
                    break;

                default:
                    if (std::isdigit(static_cast<unsigned char>(ch)) || ch == '.')
                    {
                        tokens.push_back(read_number());
                    }
                    else
                    {
                        throw std::runtime_error(
```

```
                            "Unexpected character: " + std::string(1, ch));
                    }
                    break;
            }
        }

        return tokens;
    }

private:
    Token read_number()
    {
        std::size_t start = pos_;
        bool dot_seen = false;

        while (pos_ < input_.size())
        {
            char ch = input_[pos_];

            if (std::isdigit(static_cast<unsigned char>(ch)))
            {
                ++pos_;
            }
            else if (ch == '.')
            {
                if (dot_seen)
                    break;

                dot_seen = true;
                ++pos_;
            }
            else
            {
                break;
            }
        }

        std::string text(input_.substr(start, pos_ - start));

        double value{};
        auto result = std::from_chars(
            text.data(),
            text.data() + text.size(),
            value
        );

        if (result.ec != std::errc{})
        {
            throw std::runtime_error("Invalid number: " + text);
        }

        return Token{TokenType::Number, value, text};
```

```
    }

    void skip_spaces()
    {
        while (pos_ < input_.size() &&
               std::isspace(static_cast<unsigned char>(input_[pos_])))
        {
            ++pos_;
        }
    }

private:
    std::string_view input_;
    std::size_t pos_;
};

class Parser
{
public:
    explicit Parser(std::vector<Token> tokens)
        : tokens_(std::move(tokens)), pos_(0)
    {
    }

    double parse()
    {
        double result = parse_expression();

        if (current().type != TokenType::End)
        {
            throw std::runtime_error(
                "Unexpected token after complete expression: " + current().text);
        }

        return result;
    }

private:
    double parse_expression()
    {
        double left = parse_term();

        while (current().type == TokenType::Plus ||
               current().type == TokenType::Minus)
        {
            TokenType op = current().type;
            advance();

            double right = parse_term();

            if (op == TokenType::Plus)
                left += right;
```

```
            else
                left -= right;
        }

        return left;
    }

    double parse_term()
    {
        double left = parse_factor();

        while (current().type == TokenType::Star ||
               current().type == TokenType::Slash)
        {
            TokenType op = current().type;
            advance();

            double right = parse_factor();

            if (op == TokenType::Star)
            {
                left *= right;
            }
            else
            {
                if (right == 0.0)
                    throw std::runtime_error("Division by zero");

                left /= right;
            }
        }

        return left;
    }

    double parse_factor()
    {
        if (current().type == TokenType::Minus)
        {
            advance();
            return -parse_factor();
        }

        if (current().type == TokenType::Number)
        {
            double value = current().value;
            advance();
            return value;
        }

        if (current().type == TokenType::LeftParen)
        {
```

```
            advance();
            double value = parse_expression();

            if (current().type != TokenType::RightParen)
                throw std::runtime_error("Missing closing parenthesis");

            advance();
            return value;
        }

        throw std::runtime_error("Expected number or '('");
    }

    const Token& current() const
    {
        return tokens_[pos_];
    }

    void advance()
    {
        if (pos_ < tokens_.size())
            ++pos_;
    }

private:
    std::vector<Token> tokens_;
    std::size_t pos_;
};

int main()
{
    std::cout << "Modern C++ Calculator\n";
    std::cout << "Type an expression or 'exit' to quit.\n\n";

    std::string line;

    while (true)
    {
        std::cout << "> ";

        if (!std::getline(std::cin, line))
            break;

        if (line == "exit")
            break;

        if (line.empty())
            continue;

        try
        {
            Tokenizer tokenizer(line);
```

```
            auto tokens = tokenizer.tokenize();

            Parser parser(std::move(tokens));
            double result = parser.parse();

            std::cout << "= " << result << "\n\n";
        }
        catch (const std::exception& ex)
        {
            std::cout << "Error: " << ex.what() << "\n\n";
        }
    }

    return 0;
}
```

42.11 Windows compilation

This program can be compiled on Windows using common modern compilers.

42.11.1 MSVC

```
cl /std:c++20 /EHsc /W4 calculator.cpp
```

42.11.2 Clang on Windows

```
clang++ -std=c++20 -Wall -Wextra -pedantic calculator.cpp -o calculator.exe
```

42.11.3 GCC on Windows

```
g++ -std=c++20 -Wall -Wextra -pedantic calculator.cpp -o calculator.exe
```

42.12 Sample session

```
Modern C++ Calculator
Type an expression or 'exit' to quit.

> 2 + 3 * 4
= 14

> (2 + 3) * 4
= 20

> 10 / 2 + 6
= 11

> 5 / 0
Error: Division by zero
```

```
> 3 +
Error: Expected number or '('

> exit
```

42.13 Design discussion

This calculator is small, but it already shows several important software design ideas.

42.13.1 Single responsibility

Each part of the program has a distinct role:

- the tokenizer reads characters and produces tokens,
- the parser processes tokens using grammar rules,
- the main loop handles interaction with the user.

This is better than placing all logic into one very large function.

42.13.2 Grammar-driven parsing

The recursive-descent structure mirrors the grammar:

- expression handles addition and subtraction,
- term handles multiplication and division,
- factor handles numbers, parentheses, and unary minus.

This makes the code easier to reason about because the program structure matches the mathematical structure of the language.

42.13.3 Readable error handling

When the user enters invalid input, the program reports a useful message. This is far better than silent failure or undefined behavior.
Examples:

- `Unexpected character: a`
- `Missing closing parenthesis`
- `Division by zero`
- `Expected number or '('`

A real program should always help the user understand what went wrong.

42.14 Extending the calculator

Once the base design is clean, the project becomes easy to extend.
Possible future features include:

- variables such as `x = 10`,
- exponentiation such as `2 ^ 8`,
- functions such as `sin(0.5)` or `sqrt(16)`,
- command history,
- support for integer-only mode,
- support for `std::expected` instead of exceptions,
- conversion from direct evaluation to syntax tree construction.

42.14.1 Adding exponentiation

To add exponentiation correctly, one would normally introduce another grammar level, for example:

```
expression = term { ('+' | '-') term }
term       = power { ('*' | '/') power }
power      = factor { '^' power }
factor     = number | '(' expression ')' | unary minus
```

This shows how clean grammar design makes future growth easier.

42.15 Alternative design: building an AST

The current calculator evaluates expressions immediately while parsing. That is efficient and simple, but another important design is to build an **abstract syntax tree** first.
For example, the expression:

```
2 + 3 * 4
```

could be represented as a tree:

```
    +
   / \
  2   *
     / \
    3   4
```

An AST design is useful when:

- you want to optimize expressions,
- you want to print expressions back in structured form,

- you want to interpret or compile them later,
- you want to separate parsing from evaluation completely.

For a beginner project, direct evaluation is usually the best first step. But understanding that an AST-based design exists is important.

42.16 Testing the project

A calculator should be tested with both valid and invalid inputs.

42.16.1 Basic valid tests

```
1 + 2           -> 3
2 * 3 + 4       -> 10
2 + 3 * 4       -> 14
(2 + 3) * 4     -> 20
-5 + 3          -> -2
2.5 * 4         -> 10
```

42.16.2 Basic invalid tests

```
1 +
(2 + 3
2..5
5 / 0
hello
```

42.16.3 Why testing matters

Testing here is not only about correctness. It also verifies:

- operator precedence,
- parenthesis handling,
- unary minus behavior,
- numeric parsing,
- error message quality.

Even a tiny program benefits from disciplined testing.

42.17 Common beginner mistakes

When writing a calculator project, beginners often make the following mistakes:

42.17.1 Mixing tokenization and parsing

Trying to parse directly from raw characters inside every function quickly becomes messy. Tokenization first is usually cleaner.

42.17.2 Ignoring spaces

A calculator should accept input naturally, including spaces around operators.

42.17.3 Forgetting precedence

A naive left-to-right evaluator would compute:

```
2 + 3 * 4
```

as:

```
(2 + 3) * 4 = 20
```

which is wrong for standard arithmetic interpretation.

42.17.4 Weak error handling

Programs should not crash or produce meaningless answers for bad input.

42.17.5 Using giant functions

Large monolithic functions become difficult to read, debug, and extend.

42.18 Professional lessons from this project

Although this project is small, it teaches lessons that apply far beyond calculators:

- clean input handling,
- layered design,
- structured parsing,
- reliable error reporting,
- using standard library tools wisely,
- writing code that can grow in future chapters.

This is exactly why small projects matter. They teach discipline in a manageable environment.

42.19 Possible improved version with commands

A practical extension is to support simple commands such as:

- `exit`
- `help`
- `clear`

Example design:

```
if (line == "exit")
{
    break;
}
else if (line == "help")
{
    std::cout << "Enter arithmetic expressions using +, -, *, /, and parentheses.\n";
    continue;
}
```

This small change begins to move the program from a raw exercise toward a usable interactive tool.

42.20 Refactoring ideas

Once the chapter is understood, the project can be refactored in several useful ways:

- move tokenizer and parser into separate header and source files,
- add unit tests,
- replace exception messages with richer diagnostics,
- store token positions for better error reporting,
- support AST building,
- add functions and variables.

42.21 Final complete version with token positions

A more advanced design stores source positions so the calculator can report where an error occurred.

```
struct Token
{
    TokenType type{};
    double value{};
    std::string text{};
    std::size_t position{};
};
```

Then an error such as:

```
2 + * 3
```

could be reported as:

```
Error at position 4: Expected number or '('
```

This is a valuable step toward compiler-style diagnostics.

Summary

This chapter built a real calculator project using Modern C++.
The project introduced:

- tokenization,
- recursive-descent parsing,
- operator precedence,
- parentheses handling,
- unary minus,
- interactive input,
- clean error handling,
- extensible program structure.

Even though the final program is modest in size, it teaches several core professional habits:

- separate responsibilities clearly,
- model the grammar explicitly,
- validate input,
- handle errors cleanly,
- build code that can evolve.

A calculator is therefore much more than a toy. It is a first step toward interpreters, compilers, command processors, expression engines, configuration parsers, and many other real software systems.

Project: Task Manager

43.1 Project overview

In this chapter, we build a complete console-based Task Manager in Modern C++. This project is larger and more realistic than a simple calculator because it combines several software engineering concerns in one coherent application:

- structured data modeling,
- command parsing,
- dynamic storage of records,
- file persistence,
- searching and filtering,
- input validation,
- clean separation of responsibilities.

A task manager is an excellent learning project because it resembles the kind of utility software many developers build in real life. It is small enough for a beginner to understand, but rich enough to introduce professional habits.
The program we design will allow the user to:

- add tasks,
- list tasks,
- mark tasks as done,
- delete tasks,
- search tasks,
- save tasks to disk,
- load tasks automatically when the program starts.

This chapter emphasizes code clarity, maintainability, and realistic architecture rather than writing everything inside one large function.

43.2 Project goals

The goals of this project are:

- to design a reusable task data model,
- to store tasks in a standard container,
- to support interactive text commands,
- to persist tasks in a file,
- to validate numeric input safely,
- to write code that can later grow into a GUI, web, or database-backed tool.

A good beginner project should not only work. It should teach good structure. For this reason, we will divide the application into logical layers instead of mixing all behavior together.

43.3 Features of the final program

The final version of the program will support commands such as:

```
add Write chapter 43
add Review parsing code
list
done 1
remove 2
find chapter
help
exit
```

Example output:

```
[1] [ ] Write chapter 43
[2] [x] Review parsing code
```

Each task will have:

- an integer ID,
- a title,
- a completion state,
- a creation timestamp stored as a string for simplicity in this introductory project.

43.4 Why this project matters

Many beginners first learn syntax and basic statements, but struggle when they try to combine those features into a full program. This chapter solves that problem by demonstrating how several language and library features work together.
This project teaches:

- classes and structs,
- vectors,
- strings,
- optional values,
- file input and output,
- command dispatch,
- parsing and validation,
- practical use of standard library facilities.

This is the kind of chapter that begins to transform isolated knowledge into practical programming ability.

43.5 High-level architecture

We will organize the project around the following components:

- `Task`: a data structure representing one task,
- `TaskManager`: the main application class,
- helper functions for parsing commands,
- file save and load functions,
- `main()`: the interactive loop.

This separation keeps the design clear:

- the task structure stores data,
- the manager owns the collection of tasks,
- parsing helpers interpret user commands,
- the main loop handles console interaction.

43.6 Data model

The first design step is to define what a task is.

```
#include <string>

struct Task
{
    int id{};
    std::string title;
    bool completed{};
    std::string created_at;
};
```

This model is intentionally simple.

- `id` uniquely identifies a task during the program session.
- `title` stores the text entered by the user.
- `completed` records whether the task has been finished.
- `created_at` stores a readable creation time.

A more advanced program might use a richer time representation or store priority, category, and deadline. For this learning project, clarity is more important than excessive complexity.

43.7 Choosing the container

The natural container for this project is `std::vector<Task>`.
Why is `std::vector` a good choice here?

- tasks form a sequence,
- we often iterate through all tasks,
- random access is useful,
- dynamic growth is necessary,
- the project is moderate in scale.

```
#include <vector>

std::vector<Task> tasks;
```

For a beginner-friendly task manager, `std::vector` is the most natural and practical standard container.

43.8 The TaskManager class

We now define the main class that owns the task list and provides operations on it.

```
#include <algorithm>
#include <charconv>
#include <chrono>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <optional>
#include <sstream>
#include <string>
#include <string_view>
#include <vector>

class TaskManager
{
public:
    TaskManager()
        : next_id_(1)
    {
    }

    void add_task(const std::string& title)
    {
        if (title.empty())
        {
            std::cout << "Task title cannot be empty.\n";
            return;
        }

        tasks_.push_back(Task{
            next_id_++,
            title,
            false,
            current_time_string()
        });

        std::cout << "Task added successfully.\n";
    }

    void list_tasks() const
    {
        if (tasks_.empty())
        {
            std::cout << "No tasks available.\n";
            return;
        }

        for (const auto& task : tasks_)
```

```
        {
            std::cout
                << '[' << task.id << "] "
                << '[' << (task.completed ? 'x' : ' ') << "] "
                << task.title
                << " (created: " << task.created_at << ")\n";
        }
    }

    void mark_done(int id)
    {
        auto task = find_task_by_id(id);

        if (!task)
        {
            std::cout << "Task not found.\n";
            return;
        }

        task->get().completed = true;
        std::cout << "Task marked as done.\n";
    }

    void remove_task(int id)
    {
        auto old_size = tasks_.size();

        tasks_.erase(
            std::remove_if(tasks_.begin(), tasks_.end(),
                [id](const Task& task)
                {
                    return task.id == id;
                }),
            tasks_.end()
        );

        if (tasks_.size() == old_size)
            std::cout << "Task not found.\n";
        else
            std::cout << "Task removed.\n";
    }

    void find_tasks(std::string_view keyword) const
    {
        bool found = false;

        for (const auto& task : tasks_)
        {
            if (task.title.find(keyword) != std::string::npos)
            {
                std::cout
                    << '[' << task.id << "] "
```

```
                << '[' << (task.completed ? 'x' : ' ') << "] "
                << task.title
                << " (created: " << task.created_at << ")\n";

            found = true;
        }
    }

    if (!found)
        std::cout << "No matching tasks found.\n";
}

void save_to_file(const std::filesystem::path& file_path) const
{
    std::filesystem::create_directories(file_path.parent_path());

    std::ofstream out(file_path);

    if (!out)
    {
        std::cout << "Failed to save tasks.\n";
        return;
    }

    out << next_id_ << '\n';
    out << tasks_.size() << '\n';

    for (const auto& task : tasks_)
    {
        out << task.id << '\n';
        out << task.completed << '\n';
        out << task.created_at << '\n';
        out << task.title << '\n';
    }
}

void load_from_file(const std::filesystem::path& file_path)
{
    std::ifstream in(file_path);

    if (!in)
        return;

    tasks_.clear();

    std::string line;

    if (!std::getline(in, line))
        return;

    auto next_id = parse_int(line);
    if (!next_id)
```

```
        return;

    next_id_ = *next_id;

    if (!std::getline(in, line))
        return;

    auto count = parse_int(line);
    if (!count)
        return;

    for (int i = 0; i < *count; ++i)
    {
        Task task;

        if (!std::getline(in, line))
            return;
        auto id = parse_int(line);
        if (!id)
            return;
        task.id = *id;

        if (!std::getline(in, line))
            return;
        auto completed = parse_int(line);
        if (!completed)
            return;
        task.completed = (*completed != 0);

        if (!std::getline(in, task.created_at))
            return;

        if (!std::getline(in, task.title))
            return;

        tasks_.push_back(std::move(task));
    }
}

static void print_help()
{
    std::cout
        << "Commands:\n"
        << "  add <title>      Add a new task\n"
        << "  list             Show all tasks\n"
        << "  done <id>        Mark a task as completed\n"
        << "  remove <id>      Remove a task\n"
        << "  find <keyword>   Search tasks by text\n"
        << "  help             Show this help message\n"
        << "  exit             Save and quit\n";
}
```

```
private:
    std::optional<std::reference_wrapper<Task>> find_task_by_id(int id)
    {
        for (auto& task : tasks_)
        {
            if (task.id == id)
                return task;
        }

        return std::nullopt;
    }

    static std::optional<int> parse_int(std::string_view text)
    {
        int value{};
        auto result = std::from_chars(text.data(), text.data() + text.size(), value);

        if (result.ec != std::errc{} || result.ptr != text.data() + text.size())
            return std::nullopt;

        return value;
    }

    static std::string current_time_string()
    {
        using clock = std::chrono::system_clock;

        const auto now = clock::now();
        const std::time_t current = clock::to_time_t(now);

        std::tm local_tm{};

    #ifdef _WIN32
        localtime_s(&local_tm, &current);
    #else
        local_tm = *std::localtime(&current);
    #endif

        std::ostringstream out;
        out << std::put_time(&local_tm, "%Y-%m-%d %H:%M:%S");
        return out.str();
    }

private:
    std::vector<Task> tasks_;
    int next_id_;
};
```

43.9 Why this design is good

The design above follows several important software engineering principles.

43.9.1 Encapsulation

The task list is owned by the `TaskManager` class rather than being spread across global variables. This makes the program easier to reason about and easier to extend.

43.9.2 Single responsibility

Each member function has one clear role:

- `add_task()` adds tasks,
- `list_tasks()` prints tasks,
- `mark_done()` changes completion state,
- `remove_task()` deletes tasks,
- `find_tasks()` searches text,
- `save_to_file()` writes persistent data,
- `load_from_file()` restores persistent data.

43.9.3 Safe parsing

The program uses `std::from_chars` inside `parse_int()` to parse integer command arguments. This is a good modern C++ technique because it is explicit and easy to validate.

43.9.4 Optional return values

`std::optional` is used to represent two common situations:

- an ID lookup may fail,
- integer parsing may fail.

This is much better than returning meaningless sentinel values such as `-1`.

43.10 Command parsing strategy

The program reads one full input line at a time using `std::getline`. This is the correct design for command-style tools because commands often contain spaces.
For example:

```
add Finish the task manager chapter
```

If we used `operator»` directly, the title would stop at the first space. Reading the full line avoids that problem.
We then split the line into:

- the command word,
- the remainder of the line.

43.11 Command loop

The following `main()` function runs the interactive console loop.

```
int main()
{
    const std::filesystem::path data_file = "data/tasks.txt";

    TaskManager manager;
    manager.load_from_file(data_file);

    std::cout << "Modern C++ Task Manager\n";
    std::cout << "Type 'help' to see available commands.\n\n";

    std::string line;

    while (true)
    {
        std::cout << "> ";

        if (!std::getline(std::cin, line))
            break;

        if (line.empty())
            continue;

        std::istringstream input(line);

        std::string command;
        input >> command;

        if (command == "add")
        {
            std::string title;
            std::getline(input >> std::ws, title);
            manager.add_task(title);
        }
        else if (command == "list")
        {
            manager.list_tasks();
        }
        else if (command == "done")
        {
            std::string id_text;
            input >> id_text;

            auto id = TaskManager::parse_int(id_text);

            if (!id)
                std::cout << "Invalid task ID.\n";
            else
                manager.mark_done(*id);
```

```
        }
        else if (command == "remove")
        {
            std::string id_text;
            input >> id_text;

            auto id = TaskManager::parse_int(id_text);

            if (!id)
                std::cout << "Invalid task ID.\n";
            else
                manager.remove_task(*id);
        }
        else if (command == "find")
        {
            std::string keyword;
            std::getline(input >> std::ws, keyword);

            if (keyword.empty())
                std::cout << "Please provide a search keyword.\n";
            else
                manager.find_tasks(keyword);
        }
        else if (command == "help")
        {
            TaskManager::print_help();
        }
        else if (command == "exit")
        {
            manager.save_to_file(data_file);
            std::cout << "Tasks saved. Goodbye.\n";
            break;
        }
        else
        {
            std::cout << "Unknown command. Type 'help' for guidance.\n";
        }

        std::cout << '\n';
    }

    return 0;
}
```

43.12 Small improvement for access control

The version above calls `TaskManager::parse_int()` from `main()`. To allow that cleanly, the helper should be public instead of private, or the parsing logic should be moved to a separate free function.
A clean version is:

```
public:
```

```
    static std::optional<int> parse_int(std::string_view text)
    {
        int value{};
        auto result = std::from_chars(text.data(), text.data() + text.size(), value);

        if (result.ec != std::errc{} || result.ptr != text.data() + text.size())
            return std::nullopt;

        return value;
    }
```

This is a good example of normal code review and refactoring practice. Writing a project chapter should not hide such realistic design improvements.

43.13 Complete program

Below is the full integrated program, ready to paste into one source file.

```
#include <algorithm>
#include <charconv>
#include <chrono>
#include <filesystem>
#include <fstream>
#include <functional>
#include <iomanip>
#include <iostream>
#include <optional>
#include <sstream>
#include <string>
#include <string_view>
#include <vector>

struct Task
{
    int id{};
    std::string title;
    bool completed{};
    std::string created_at;
};

class TaskManager
{
public:
    TaskManager()
        : next_id_(1)
    {
    }

    void add_task(const std::string& title)
    {
        if (title.empty())
```

```
    {
        std::cout << "Task title cannot be empty.\n";
        return;
    }

    tasks_.push_back(Task{
        next_id_++,
        title,
        false,
        current_time_string()
    });

    std::cout << "Task added successfully.\n";
}

void list_tasks() const
{
    if (tasks_.empty())
    {
        std::cout << "No tasks available.\n";
        return;
    }

    for (const auto& task : tasks_)
    {
        std::cout
            << '[' << task.id << "] "
            << '[' << (task.completed ? 'x' : ' ') << "] "
            << task.title
            << " (created: " << task.created_at << ")\n";
    }
}

void mark_done(int id)
{
    auto task = find_task_by_id(id);

    if (!task)
    {
        std::cout << "Task not found.\n";
        return;
    }

    task->get().completed = true;
    std::cout << "Task marked as done.\n";
}

void remove_task(int id)
{
    auto old_size = tasks_.size();

    tasks_.erase(
```

```
        std::remove_if(tasks_.begin(), tasks_.end(),
            [id](const Task& task)
            {
                return task.id == id;
            }),
        tasks_.end()
    );

    if (tasks_.size() == old_size)
        std::cout << "Task not found.\n";
    else
        std::cout << "Task removed.\n";
}

void find_tasks(std::string_view keyword) const
{
    bool found = false;

    for (const auto& task : tasks_)
    {
        if (task.title.find(keyword) != std::string::npos)
        {
            std::cout
                << '[' << task.id << "] "
                << '[' << (task.completed ? 'x' : ' ') << "] "
                << task.title
                << " (created: " << task.created_at << ")\n";

            found = true;
        }
    }

    if (!found)
        std::cout << "No matching tasks found.\n";
}

void save_to_file(const std::filesystem::path& file_path) const
{
    std::filesystem::create_directories(file_path.parent_path());

    std::ofstream out(file_path);

    if (!out)
    {
        std::cout << "Failed to save tasks.\n";
        return;
    }

    out << next_id_ << '\n';
    out << tasks_.size() << '\n';

    for (const auto& task : tasks_)
```

```
        {
            out << task.id << '\n';
            out << task.completed << '\n';
            out << task.created_at << '\n';
            out << task.title << '\n';
        }
    }

    void load_from_file(const std::filesystem::path& file_path)
    {
        std::ifstream in(file_path);

        if (!in)
            return;

        tasks_.clear();

        std::string line;

        if (!std::getline(in, line))
            return;

        auto next_id = parse_int(line);
        if (!next_id)
            return;

        next_id_ = *next_id;

        if (!std::getline(in, line))
            return;

        auto count = parse_int(line);
        if (!count)
            return;

        for (int i = 0; i < *count; ++i)
        {
            Task task;

            if (!std::getline(in, line))
                return;
            auto id = parse_int(line);
            if (!id)
                return;
            task.id = *id;

            if (!std::getline(in, line))
                return;
            auto completed = parse_int(line);
            if (!completed)
                return;
            task.completed = (*completed != 0);
```

```
            if (!std::getline(in, task.created_at))
                return;

            if (!std::getline(in, task.title))
                return;

            tasks_.push_back(std::move(task));
        }
    }

    static void print_help()
    {
        std::cout
            << "Commands:\n"
            << "  add <title>      Add a new task\n"
            << "  list             Show all tasks\n"
            << "  done <id>        Mark a task as completed\n"
            << "  remove <id>      Remove a task\n"
            << "  find <keyword>   Search tasks by text\n"
            << "  help             Show this help message\n"
            << "  exit             Save and quit\n";
    }

    static std::optional<int> parse_int(std::string_view text)
    {
        int value{};
        auto result = std::from_chars(text.data(), text.data() + text.size(), value);

        if (result.ec != std::errc{} || result.ptr != text.data() + text.size())
            return std::nullopt;

        return value;
    }

private:
    std::optional<std::reference_wrapper<Task>> find_task_by_id(int id)
    {
        for (auto& task : tasks_)
        {
            if (task.id == id)
                return task;
        }

        return std::nullopt;
    }

    static std::string current_time_string()
    {
        using clock = std::chrono::system_clock;

        const auto now = clock::now();
```

```
        const std::time_t current = clock::to_time_t(now);

        std::tm local_tm{};

    #ifdef _WIN32
        localtime_s(&local_tm, &current);
    #else
        local_tm = *std::localtime(&current);
    #endif

        std::ostringstream out;
        out << std::put_time(&local_tm, "%Y-%m-%d %H:%M:%S");
        return out.str();
    }

private:
    std::vector<Task> tasks_;
    int next_id_;
};

int main()
{
    const std::filesystem::path data_file = "data/tasks.txt";

    TaskManager manager;
    manager.load_from_file(data_file);

    std::cout << "Modern C++ Task Manager\n";
    std::cout << "Type 'help' to see available commands.\n\n";

    std::string line;

    while (true)
    {
        std::cout << "> ";

        if (!std::getline(std::cin, line))
            break;

        if (line.empty())
            continue;

        std::istringstream input(line);

        std::string command;
        input >> command;

        if (command == "add")
        {
            std::string title;
            std::getline(input >> std::ws, title);
            manager.add_task(title);
```

```
        }
        else if (command == "list")
        {
            manager.list_tasks();
        }
        else if (command == "done")
        {
            std::string id_text;
            input >> id_text;

            auto id = TaskManager::parse_int(id_text);

            if (!id)
                std::cout << "Invalid task ID.\n";
            else
                manager.mark_done(*id);
        }
        else if (command == "remove")
        {
            std::string id_text;
            input >> id_text;

            auto id = TaskManager::parse_int(id_text);

            if (!id)
                std::cout << "Invalid task ID.\n";
            else
                manager.remove_task(*id);
        }
        else if (command == "find")
        {
            std::string keyword;
            std::getline(input >> std::ws, keyword);

            if (keyword.empty())
                std::cout << "Please provide a search keyword.\n";
            else
                manager.find_tasks(keyword);
        }
        else if (command == "help")
        {
            TaskManager::print_help();
        }
        else if (command == "exit")
        {
            manager.save_to_file(data_file);
            std::cout << "Tasks saved. Goodbye.\n";
            break;
        }
        else
        {
            std::cout << "Unknown command. Type 'help' for guidance.\n";
```

```
        }

        std::cout << '\n';
    }

    return 0;
}
```

43.14 How persistence works

The program stores data in a simple text file:

```
4
3
1
0
2026-03-20 12:00:01
Write chapter 43
2
1
2026-03-20 12:05:13
Review task manager design
3
0
2026-03-20 12:07:54
Test file loading
```

The format is intentionally straightforward:

- first line: next available ID,
- second line: number of tasks,
- then four lines per task:
  - ID,
  - completion flag,
  - creation time,
  - title.

This is not the most advanced persistence format, but it is easy to understand and easy to debug.

43.15 Why text persistence is useful here

For a beginner-focused project, text persistence has several advantages:

- it is readable,
- it is easy to inspect manually,

- it avoids binary-format complexity,
- it reinforces string and stream skills,
- it is sufficient for a small console tool.

A later version could migrate to JSON, SQLite, or a network service, but that is beyond the scope of this chapter.

43.16 Searching and filtering

The `find_tasks()` member performs a simple substring search:

```
if (task.title.find(keyword) != std::string::npos)
```

This is a good first step. It is easy to understand and works well enough for a small tool.
A more advanced version might support:

- case-insensitive matching,
- regular expressions,
- filtering by completed state,
- filtering by date,
- sorting results.

43.17 Command examples

Here is a sample interaction:

```
Modern C++ Task Manager
Type 'help' to see available commands.

> add Write chapter 43
Task added successfully.

> add Review saved-file format
Task added successfully.

> list
[1] [ ] Write chapter 43 (created: 2026-03-20 18:30:10)
[2] [ ] Review saved-file format (created: 2026-03-20 18:30:20)

> done 1
Task marked as done.

> list
[1] [x] Write chapter 43 (created: 2026-03-20 18:30:10)
[2] [ ] Review saved-file format (created: 2026-03-20 18:30:20)
```

```
> find chapter
[1] [x] Write chapter 43 (created: 2026-03-20 18:30:10)

> remove 2
Task removed.

> exit
Tasks saved. Goodbye.
```

43.18 Windows compilation

This program is suitable for common Windows toolchains.

43.18.1 MSVC

```
cl /std:c++20 /EHsc /W4 task_manager.cpp
```

43.18.2 Clang on Windows

```
clang++ -std=c++20 -Wall -Wextra -pedantic task_manager.cpp -o task_manager.exe
```

43.18.3 GCC on Windows

```
g++ -std=c++20 -Wall -Wextra -pedantic task_manager.cpp -o task_manager.exe
```

43.19 Testing ideas

A real project should always be tested. For this task manager, useful manual tests include:

43.19.1 Add command tests

- add a normal task,
- add an empty task,
- add a long task title,
- add a title with spaces.

43.19.2 Done and remove tests

- mark a valid ID as done,
- mark an invalid ID as done,
- remove a valid ID,
- remove an invalid ID.

43.19.3 Persistence tests

- add tasks and exit,
- restart the program and verify the tasks were restored,
- verify completed state is preserved,
- verify next ID continues correctly.

43.19.4 Search tests

- search for existing text,
- search for nonexisting text,
- search using spaces,
- search with empty input.

43.20 Common beginner mistakes

A task manager project often reveals several common programming mistakes.

43.20.1 Using global variables everywhere

This makes the project hard to extend and harder to test. Keeping the task collection inside `TaskManager` is cleaner.

43.20.2 Parsing input with only `operator»`

That approach usually loses spaces in task titles. Reading the whole line first is much better.

43.20.3 Returning invalid sentinel values

For example, using `-1` for a missing ID or a failed parse is weak design. `std::optional` communicates failure explicitly.

43.20.4 Mixing file format and business logic everywhere

When save and load logic are isolated in dedicated functions, the rest of the program stays cleaner.

43.20.5 Ignoring invalid input

Professional software must handle bad input gracefully.

43.21 Professional lessons from this project

This project introduces several habits that remain important in much larger systems.

43.21.1 Design your data first

Before writing many functions, define the shape of the data. A clear `Task` structure made the rest of the program easier to build.

43.21.2 Choose standard abstractions well

This chapter demonstrates that a surprisingly useful program can be built with ordinary standard components:

- `std::vector`,
- `std::string`,
- `std::optional`,
- `std::filesystem`,
- file streams,
- character conversion,
- time formatting.

43.21.3 Prefer small focused functions

Small functions are easier to debug, reason about, and improve.

43.21.4 Build for extension

Even though this is a console project, the core logic is structured well enough that it could later support:

- a GUI front end,
- a REST backend,
- a JSON storage layer,
- task priorities,
- categories,
- deadlines,
- sorting and filtering commands.

43.22 Possible extensions

Here are realistic next steps for improving the task manager.

43.22.1 Task priorities

Add a priority field:

```
enum class Priority
{
    Low,
    Medium,
    High
};
```

Then store it inside `Task` and allow commands such as:

```
add high Finish chapter
```

43.22.2 Sorting tasks

A sort command could arrange tasks by ID, title, or completion state.

```
std::sort(tasks_.begin(), tasks_.end(),
    [](const Task& a, const Task& b)
    {
        return a.title < b.title;
    });
```

43.22.3 Separate completed and active tasks

A richer `list` command might show unfinished tasks first.

43.22.4 Better persistence format

Later chapters or projects could replace the custom text format with:

- JSON,
- CSV,
- SQLite,
- a client-server API.

43.22.5 Using `std::expected`

A more advanced version could replace message-based error printing with explicit success-or-error results.

43.23 Refactoring opportunities

After understanding the single-file version, a student should refactor the project into multiple files.
A good layout is:

```
task_manager/
    main.cpp
    task.hpp
    task_manager.hpp
    task_manager.cpp
```

This teaches compilation units, headers, and cleaner project organization.

43.24 A better command parsing direction

The current command parser is simple and good for beginners, but in a larger program you might eventually introduce:

- a command structure,
- tokenized command arguments,
- quoted-string support,
- dedicated parser functions per command.

That evolution mirrors how real command-line tools grow over time.

Summary

This chapter built a complete Task Manager in Modern C++.
The project demonstrated:

- a clear data model,
- container-based task storage,
- optional-based lookup and parsing,
- interactive command handling,
- text-file persistence,
- simple search,
- Windows-friendly compilation,
- realistic extension paths.

More importantly, it showed how a real application emerges from combining basic language features with the standard library in a disciplined way.
A task manager may look like a small project, but it teaches several core professional ideas:

- organize code by responsibility,
- validate input carefully,

- design clean data structures,
- persist state safely,
- write code that can evolve.

That is exactly what real-world Modern C++ programming requires.

Project: Data Library

44.1 Project overview

In this chapter, we design and implement a reusable Data Library in Modern C++. Unlike previous projects (Calculator and Task Manager), this project is not an application but a **library** that can be reused in multiple programs.
The goal is to build a small but powerful framework for:

- storing structured data,
- loading data from files,
- saving data to files,
- querying and filtering data,
- designing clean reusable APIs.

This project reflects real-world software development, where libraries are built once and reused many times.

44.2 Motivation

Modern applications rely heavily on structured data:

- configuration systems,
- databases,
- analytics tools,
- backend services,
- scientific computing systems.

C++ provides powerful tools for handling structured data using:

- containers such as `std::vector`,
- file I/O using `std::ifstream` and `std::ofstream`,
- parsing and transformation logic,

- strong type systems.

The standard library offers generic containers and algorithms that make data handling efficient and flexible.
This project teaches how to combine these features into a reusable system.

44.3 Design goals

The Data Library should:

- be generic and reusable,
- separate data storage from logic,
- support file persistence,
- allow querying and filtering,
- be easy to extend.

We will implement:

- a data record structure,
- a container-based storage,
- a manager class,
- file parsing logic,
- query functions.

44.4 Data model

We define a simple record representing a row of structured data.

```
#include <string>

struct Record
{
    int id{};
    std::string name;
    double value{};
};
```

This represents a generic dataset such as:

- product lists,
- sensor readings,
- configuration entries,
- financial data.

44.5 Storage container

We use `std::vector<Record>` as the main storage.
This is appropriate because:

- data is sequential,
- dynamic resizing is required,
- random access is efficient,
- iteration is frequent.

Vectors store elements contiguously and provide fast access and efficient memory usage.

```
#include <vector>

std::vector<Record> records;
```

44.6 DataLibrary class

We now define a reusable class to manage records.

```
#include <algorithm>
#include <charconv>
#include <fstream>
#include <iostream>
#include <optional>
#include <sstream>
#include <string>
#include <string_view>
#include <vector>

class DataLibrary
{
public:
    void add_record(const Record& r)
    {
        records_.push_back(r);
    }

    void list() const
    {
        for (const auto& r : records_)
        {
            std::cout << r.id << " | "
                      << r.name << " | "
                      << r.value << "\n";
        }
    }
```

```
    std::optional<Record> find_by_id(int id) const
    {
        for (const auto& r : records_)
        {
            if (r.id == id)
                return r;
        }
        return std::nullopt;
    }

    std::vector<Record> filter_by_value(double min_value) const
    {
        std::vector<Record> result;

        for (const auto& r : records_)
        {
            if (r.value >= min_value)
                result.push_back(r);
        }

        return result;
    }
```

44.7 File loading

Reading structured data from files is a core skill. C++ uses `std::ifstream` for file input.

```
    bool load_from_file(const std::string& filename)
    {
        std::ifstream file(filename);

        if (!file)
        {
            std::cout << "Failed to open file\n";
            return false;
        }

        records_.clear();

        std::string line;

        while (std::getline(file, line))
        {
            std::istringstream ss(line);

            Record r;

            ss >> r.id >> r.name >> r.value;

            if (ss)
                records_.push_back(r);
        }
```

```
    return true;
}
```

File parsing is commonly implemented using streams and containers, where rows are read and stored into structured data like vectors.

44.8 File saving

```
bool save_to_file(const std::string& filename) const
{
    std::ofstream file(filename);

    if (!file)
    {
        std::cout << "Failed to write file\n";
        return false;
    }

    for (const auto& r : records_)
    {
        file << r.id << " "
             << r.name << " "
             << r.value << "\n";
    }

    return true;
}
```

This simple text format is:

- easy to read,
- easy to debug,
- portable across systems.

44.9 Parsing utilities

We add safe numeric parsing using `std::from_chars`.

```
static std::optional<int> parse_int(std::string_view text)
{
    int value{};
    auto result = std::from_chars(text.data(),
                                  text.data() + text.size(),
                                  value);

    if (result.ec != std::errc{})
        return std::nullopt;
```

```
        return value;
    }
```

This avoids unsafe conversions and gives precise control over parsing.

44.10 Complete class

```
private:
    std::vector<Record> records_;
};
```

44.11 Using the library

```
int main()
{
    DataLibrary lib;

    lib.add_record({1, "A", 10.5});
    lib.add_record({2, "B", 20.0});

    lib.list();

    lib.save_to_file("data.txt");

    DataLibrary lib2;
    lib2.load_from_file("data.txt");

    lib2.list();
}
```

44.12 Example file

```
1 A 10.5
2 B 20.0
3 C 15.75
```

44.13 Query examples

```
auto result = lib.filter_by_value(15.0);

for (const auto& r : result)
{
    std::cout << r.name << "\n";
}
```

44.14 Design principles

44.14.1 Separation of concerns

- data structure: `Record`
- storage: `std::vector`
- logic: `DataLibrary`
- I/O: file functions

44.14.2 RAII usage

File streams automatically close when destroyed, preventing resource leaks.
RAII eliminates common resource management errors such as forgetting to close files.

44.14.3 Strong typing

Using a struct instead of raw arrays ensures:

- readability,
- correctness,
- maintainability.

44.15 Extending the library

This project can evolve significantly.

44.15.1 Sorting

```
std::sort(records_.begin(), records_.end(),
    [](const Record& a, const Record& b)
    {
        return a.value < b.value;
    });
```

44.15.2 Binary serialization

Instead of text:

- faster loading,
- smaller size,
- more complex implementation.

44.15.3 JSON support

Future versions could use:

- manual parsing,
- third-party libraries,
- reflection-based serialization.

Modern C++ libraries increasingly support serialization of structured data into formats such as JSON, CSV, and binary formats.

44.16 Real-world connection

Large systems use similar ideas:

- scientific frameworks store structured data efficiently,
- analytics systems use columnar data layouts,
- databases separate logical data from physical storage.

Advanced systems like ROOT demonstrate how structured C++ data can be stored and processed efficiently at scale.

44.17 Windows compilation

44.17.1 MSVC

```
cl /std:c++20 /EHsc /W4 data_library.cpp
```

44.17.2 Clang

```
clang++ -std=c++20 data_library.cpp -o data_library.exe
```

44.17.3 GCC

```
g++ -std=c++20 data_library.cpp -o data_library.exe
```

44.18 Common mistakes

- not checking file open success,
- mixing parsing logic with business logic,
- using global variables,
- unsafe numeric conversions,
- ignoring invalid input.

44.19 Professional lessons

This project introduces key engineering practices:

- building reusable libraries,
- designing clean APIs,
- separating concerns,
- using standard containers effectively,
- handling data safely and predictably.

Summary

In this chapter, we built a reusable Data Library in Modern C++.

- structured data using `struct`,
- dynamic storage using `std::vector`,
- file I/O using streams,
- safe parsing using `std::optional`,
- querying and filtering logic,
- extensible architecture.

This project marks a major transition:

- from writing programs,
- to designing reusable systems.

That transition is one of the most important steps in becoming a professional C++ developer.

Project: Multi-Module Application

45.1 Project overview

In this chapter, we build a complete multi-module application in Modern C++. Unlike previous projects, which were implemented in a single source file, this project demonstrates how to structure a real-world application across multiple compilation units.
This is a critical step toward professional development. Large systems are never written in a single file. They are composed of:

- multiple source files,
- multiple headers,
- clearly separated responsibilities,
- well-defined interfaces,
- independent compilation units.

We will transform a simple application into a structured system using:

- headers (`.hpp`),
- implementation files (`.cpp`),
- optional use of C++20 modules,
- build commands on Windows.

45.2 Motivation

Single-file programs quickly become unmanageable as complexity grows:

- code becomes difficult to navigate,
- compile times increase,
- dependencies become tangled,
- reuse becomes difficult.

Multi-module design solves these problems by:

- isolating components,
- enabling parallel development,
- reducing rebuild cost,
- improving readability.

Modern C++ encourages modular design through both traditional headers and the newer module system introduced in C++20.

45.3 Project goal

We will build a small system composed of:

- a data model module,
- a service layer,
- a storage layer,
- a user interface layer,
- a main entry point.

This mirrors real-world architecture:

- data representation,
- business logic,
- persistence,
- interaction layer.

45.4 Project structure

The project layout:

```
multi_module_app/
    main.cpp
    record.hpp
    record.cpp
    database.hpp
    database.cpp
    storage.hpp
    storage.cpp
```

Each file has a clear responsibility.

45.5 The data model

45.5.1 record.hpp

```
#pragma once

#include <string>

struct Record
{
    int id{};
    std::string name;
    double value{};
};
```

45.5.2 record.cpp

```
#include "record.hpp"
```

The implementation file is minimal because this structure is simple. In larger systems, methods and validation logic may be added.

45.6 Database module

The database manages in-memory storage.

45.6.1 database.hpp

```
#pragma once

#include "record.hpp"
#include <optional>
#include <vector>

class Database
{
public:
    void add(const Record& r);
    std::optional<Record> find(int id) const;
    void remove(int id);
    const std::vector<Record>& all() const;

private:
    std::vector<Record> records_;
};
```

45.6.2 database.cpp

```
#include "database.hpp"
#include <algorithm>
```

```
void Database::add(const Record& r)
{
    records_.push_back(r);
}

std::optional<Record> Database::find(int id) const
{
    for (const auto& r : records_)
    {
        if (r.id == id)
            return r;
    }
    return std::nullopt;
}

void Database::remove(int id)
{
    records_.erase(
        std::remove_if(records_.begin(), records_.end(),
            [id](const Record& r)
            {
                return r.id == id;
            }),
        records_.end()
    );
}

const std::vector<Record>& Database::all() const
{
    return records_;
}
```

45.7 Storage module

Handles persistence.

45.7.1 storage.hpp

```
#pragma once

#include "record.hpp"
#include <string>
#include <vector>

class Storage
{
public:
    static bool save(const std::string& file,
                     const std::vector<Record>& data);
```

```
    static bool load(const std::string& file,
                     std::vector<Record>& data);
};
```

45.7.2 storage.cpp

```
#include "storage.hpp"
#include <fstream>
#include <sstream>

bool Storage::save(const std::string& file,
                   const std::vector<Record>& data)
{
    std::ofstream out(file);

    if (!out)
        return false;

    for (const auto& r : data)
    {
        out << r.id << " "
            << r.name << " "
            << r.value << "\n";
    }

    return true;
}

bool Storage::load(const std::string& file,
                   std::vector<Record>& data)
{
    std::ifstream in(file);

    if (!in)
        return false;

    data.clear();

    std::string line;

    while (std::getline(in, line))
    {
        std::istringstream ss(line);

        Record r;

        ss >> r.id >> r.name >> r.value;

        if (ss)
            data.push_back(r);
    }
```

```
    return true;
}
```

45.8 Main application

45.8.1 main.cpp

```
#include "database.hpp"
#include "storage.hpp"
#include <iostream>
#include <string>

int main()
{
    Database db;

    Storage::load("data.txt", const_cast<std::vector<Record>&>(db.all()));

    while (true)
    {
        std::cout << "1. Add\n2. List\n3. Remove\n4. Save\n5. Exit\n> ";

        int choice{};
        std::cin >> choice;

        if (choice == 1)
        {
            Record r;

            std::cout << "ID: ";
            std::cin >> r.id;

            std::cout << "Name: ";
            std::cin >> r.name;

            std::cout << "Value: ";
            std::cin >> r.value;

            db.add(r);
        }
        else if (choice == 2)
        {
            for (const auto& r : db.all())
            {
                std::cout << r.id << " "
                          << r.name << " "
                          << r.value << "\n";
            }
        }
        else if (choice == 3)
        {
```

```
            int id;
            std::cin >> id;
            db.remove(id);
        }
        else if (choice == 4)
        {
            Storage::save("data.txt", db.all());
        }
        else if (choice == 5)
        {
            break;
        }
    }

    return 0;
}
```

45.9 Improvement: safer loading interface

The use of `const_cast` is not recommended. A better design exposes mutable access.

```
std::vector<Record>& data()
{
    return records_;
}
```

Then:

```
Storage::load("data.txt", db.data());
```

This avoids unsafe casting and follows proper encapsulation.

45.10 Compilation on Windows

45.10.1 MSVC

```
cl /std:c++20 main.cpp database.cpp storage.cpp record.cpp
```

45.10.2 Clang

```
clang++ -std=c++20 main.cpp database.cpp storage.cpp record.cpp -o app.exe
```

45.10.3 GCC

```
g++ -std=c++20 main.cpp database.cpp storage.cpp record.cpp -o app.exe
```

45.11 Transition to C++20 modules

The same project can be rewritten using modules.

45.11.1 record.ixx

```
export module record;

export struct Record
{
    int id{};
    std::string name;
    double value{};
};
```

45.11.2 database.ixx

```
export module database;

import record;
import <vector>;
import <optional>;

export class Database
{
    // same interface
};
```

Modules eliminate:

- include guards,
- macro pollution,
- redundant parsing of headers.

45.12 Benefits of multi-module design

45.12.1 Compile-time improvements

Only modified files are recompiled.

45.12.2 Encapsulation

Implementation details are hidden in `.cpp` files.

45.12.3 Scalability

Projects can grow without becoming unmanageable.

45.12.4 Team development

Multiple developers can work on different modules independently.

45.13 Common mistakes

- placing all logic in headers,
- circular dependencies,
- overusing global variables,
- exposing internal data unnecessarily,
- unsafe casting.

45.14 Professional practices

45.14.1 Header design

Headers should:

- declare interfaces,
- avoid heavy includes,
- use forward declarations when possible.

45.14.2 Source files

Source files should:

- contain implementations,
- include corresponding headers first,
- minimize dependencies.

45.14.3 Naming conventions

- `.hpp` for headers,
- `.cpp` for implementations,
- consistent file naming.

45.15 Extension ideas

- add sorting and filtering modules,
- introduce logging module,
- add configuration system,
- integrate with database systems,
- split into static or shared libraries.

Summary

This chapter demonstrated how to build a multi-module application in Modern C++.

- separated data, logic, and storage,
- implemented reusable components,
- compiled multiple translation units,
- introduced modules as a modern alternative.

This is a major milestone:

- moving from small programs,
- to structured software systems.

Understanding multi-module design is essential for any professional C++ developer.

Part XIII

Becoming Professional

Reading Official Documentation

46.1 Why official documentation matters

One of the clearest differences between a beginner and a professional C++ developer is the ability to read official documentation correctly.
A beginner often depends on scattered examples, random blog posts, copied snippets, and partial explanations from forums or videos. Those sources can sometimes be helpful, but they are not always complete, current, or correct. A professional programmer eventually learns that when correctness matters, the final authority must come from official and primary sources.
In modern C++, official documentation is important for several reasons:

- the language evolves continuously,
- compilers differ in support level,
- standard library implementations differ in maturity,
- build systems and toolchains have exact rules,
- examples found on the internet may be outdated,
- subtle wording often changes the meaning of a feature.

For example, a feature may be standardized in C++20 or C++23, but still require:

- a specific compiler version,
- a specific standard mode such as `/std:c++20` or `-std=c++23`,
- a matching standard library version,
- a platform-specific note,
- a build-system adjustment.

Only official documentation reliably tells you those details.
Reading official documentation is therefore not an optional academic skill. It is part of professional survival in C++.

46.2 What counts as official documentation

A professional C++ developer must know which sources are primary and which are secondary.
Official documentation usually includes:

- compiler vendor documentation,
- standard library vendor documentation,
- build-system manuals,
- tool manuals,
- project-maintainer documentation for major libraries,
- standards support and feature status pages.

In practical daily work, trusted official sources often include:

- Microsoft C++ documentation for MSVC, the STL, language modes, and Windows-oriented usage,
- GCC documentation and standards-support pages,
- Clang and LLVM documentation,
- CMake documentation,
- official documentation of libraries such as Boost, Qt, or other maintained toolchains.

These sources are primary because they are written and maintained by the people who build, ship, or specify the tools.
A secondary source may still be useful, but it should not overrule primary documentation.

46.3 The professional reading mindset

Many beginners open documentation and feel overwhelmed because they expect it to read like a tutorial. Official documentation is usually not written as a tutorial. It is written as a reference or technical manual.
That means you should not approach it like a storybook. You should approach it like a system.
A professional reading mindset includes the following habits:

- first identify exactly what question you are trying to answer,
- determine whether the question is about language, library, compiler, build system, platform, or version support,
- locate the correct official source,
- verify the exact version or standard mode,
- read the signature, requirements, notes, and examples carefully,
- test the result in a minimal program.

A common mistake is to search for the name of a feature and stop at the first example found online. A better workflow is:

1. identify the feature,
2. find the official documentation,
3. verify version support,
4. test with a minimal example,
5. only then integrate it into a larger project.

This habit saves enormous amounts of time.

46.4 The main categories of official documentation in C++

In real work, documentation questions usually fall into one of several categories.

46.4.1 Language documentation

Language documentation answers questions such as:

- what does `constexpr` allow,
- how do modules behave,
- what are the rules for overload resolution,
- how do coroutines work,
- what standard added this feature,
- what is the correct syntax for concepts or requires clauses.

For language support in practice, developers often rely on official compiler documentation and status pages because they show what is actually implemented in shipping tools.

46.4.2 Standard library documentation

This covers:

- containers,
- algorithms,
- iterators,
- ranges,
- formatting,
- threads and atomics,

- filesystem,
- memory utilities,
- smart pointers,
- optional and expected,
- chrono,
- span and mdspan.

When using a standard library feature, you must often verify:

- the exact header,
- required namespace,
- overload set,
- complexity and semantics,
- exception behavior,
- version availability,
- implementation notes.

46.4.3 Compiler documentation

Compiler documentation answers questions such as:

- which command-line option enables C++23 or C++26 mode,
- what warning flag produces a certain diagnostic,
- how modules are compiled,
- what ABI or code-generation option exists,
- how a specific extension behaves,
- how sanitizers or static analysis are enabled.

46.4.4 Build-system documentation

This is essential for professional work. It includes:

- how to define targets,
- how to link libraries,
- how to set include directories,

- how to export packages,
- how to configure install rules,
- how to manage generator expressions,
- how to use imported and interface targets.

A large number of C++ problems are not language problems at all. They are build-system problems.

46.4.5 Library-project documentation

When using a third-party library, the official project documentation should be the first place you consult.
For example, a professional should prefer the maintained documentation of the actual project instead of guessing from random examples.

46.5 How to read a documentation page correctly

A documentation page usually contains more information than beginners first notice.
When reading a page, pay attention to the following parts.

46.5.1 The title and scope

First ask: what exactly is this page about?
For example, is it describing:

- a class,
- a function,
- a compiler option,
- a tutorial step,
- a status table,
- a command,
- a complete subsystem.

You must know the scope before interpreting details.

46.5.2 The version context

Always identify:

- compiler version,
- tool version,

- library version,
- standard version,
- platform scope.

A page that describes a feature in current preview mode is not the same as a page describing stable production availability.

46.5.3 The required header or module

For library features, confirm:

- which header must be included,
- whether the feature belongs to `std`,
- whether special headers such as `<format>`, `<expected>`, or `<mdspan>` are required,
- whether a module import is documented.

A surprising amount of wasted debugging time comes from missing one header or using the wrong one.

46.5.4 The exact function signature

Never rely only on an example. Read the exact signature.
For example:

- are parameters references or values,
- is the function `const`,
- does it return an iterator, a boolean, a range, or an object,
- is it `constexpr`,
- is it `noexcept`,
- does it use templates or concepts.

Many misunderstandings disappear as soon as you read the full signature carefully.

46.5.5 Preconditions and requirements

Professional developers pay close attention to what must be true before a function or class is used.
Examples:

- a range must be valid,
- iterators must refer to the same sequence,
- memory must remain alive,

- an object must not be moved-from before access,
- a command must be called only after another command,
- a target must already exist in the build system.

These requirements are often more important than the main description.

46.5.6 Remarks and notes

Beginners often skip notes. Professionals do not.
A note may tell you:

- support is partial,
- behavior differs on Windows and Linux,
- a feature is experimental,
- a flag is deprecated,
- a special build order is required,
- a command should be preferred over an older alternative.

46.5.7 Examples

Examples are useful, but they should be read critically.
Ask:

- what exactly does the example prove,
- what assumptions are hidden,
- what has been omitted for brevity,
- does the example show a minimal case or a production-ready pattern,
- which parts are essential and which parts are incidental.

46.6 Reading standard library documentation professionally

When reading standard library documentation, use a checklist.

46.6.1 Checklist for a library feature

For any library facility, verify:

- the correct header,
- the exact namespace,
- template parameters,
- return type,
- whether the facility owns resources or only views them,
- iterator invalidation rules when relevant,
- complexity,
- exception guarantees,
- whether the feature is C++20, C++23, or later,
- whether your current toolchain supports it.

46.6.2 Example workflow: studying `std::span`

A professional reading process for `std::span` would look like this:

1. confirm it is a non-owning view,
2. confirm it refers to contiguous storage,
3. verify the header `<span>`,
4. read the difference between fixed extent and dynamic extent,
5. verify what constructors exist,
6. study subview operations such as `first()`, `last()`, and `subspan()`,
7. confirm lifetime rules: the underlying data must outlive the span,
8. test it with arrays, `std::array`, and `std::vector`.

46.6.3 Example workflow: studying `std::expected`

When reading about `std::expected`, a professional asks:

- what header provides it,
- what standard introduced it,
- how value and error are represented,

- how to construct unexpected errors,
- how to inspect success or failure,
- what monadic operations are available,
- whether the local standard library version fully supports them.

This is much better than memorizing one example and assuming the rest.

46.7 Reading compiler documentation professionally

Compiler documentation is one of the most important official sources in modern C++.

46.7.1 What compiler docs are for

Use compiler docs to answer questions such as:

- how to enable a standard mode,
- which warning or error flags exist,
- how to compile modules,
- how feature support differs by version,
- which extensions are available,
- how diagnostics and analysis tools work.

46.7.2 Reading a command-line option page

When reading compiler-option documentation, verify:

- what the option does,
- whether it affects parsing, code generation, linking, or diagnostics,
- whether it is language-specific,
- whether it is portable across compilers,
- whether related options must also be used.

For example, a beginner may see one command online and copy it blindly. A professional instead asks:

- is this option for GCC only,
- is there an MSVC equivalent,
- is it a stable option or experimental,
- does it require a newer release,
- does it change ABI or runtime assumptions.

46.7.3 Reading feature-status pages

Feature-status pages are extremely valuable because they answer a practical question:

What does this compiler actually support right now?

A status page is usually more useful than a general article because it shows implementation reality rather than language theory. When reading such a page, pay attention to:

- the standard version,
- whether support is full, partial, or experimental,
- the first compiler version that supports the feature,
- whether a special flag is required.

46.8 Reading build-system documentation professionally

A great many C++ developers spend years learning the language but still struggle with build systems. Professional work requires both.

46.8.1 Why build docs matter

Even if your code is correct, the project can still fail because of:

- wrong include paths,
- incorrect link order,
- missing target dependencies,
- misuse of public versus private usage requirements,
- wrong compile features,
- wrong installation rules.

46.8.2 How to read a CMake command page

Suppose you are reading documentation for a CMake command such as `add_library()` or `target_include_directories()`. A professional should verify:

- the exact syntax,
- required argument order,
- target preconditions,
- the meaning of `PRIVATE`, `PUBLIC`, and `INTERFACE`,
- whether the command modifies build requirements or usage requirements,
- how the command behaves for install or export scenarios.

46.8.3 A common professional habit

Do not memorize CMake by guessing. Read the command page whenever a command matters.
That habit avoids a large number of subtle build problems.

46.9 How to turn official docs into working knowledge

Reading documentation is only the first half. The second half is turning it into tested understanding.
A professional workflow is:

1. read the official page,
2. write the smallest possible compilable example,
3. verify the output or behavior,
4. test one failure case,
5. test one edge case,
6. only then move the feature into production code.

46.9.1 Example: testing a library feature

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

std::expected<int, std::string> parse_positive(int x)
{
    if (x > 0)
        return x;

    return std::unexpected("value must be positive");
}

int main()
{
    auto a = parse_positive(5);
    auto b = parse_positive(-1);

    if (a)
        std::cout << "a = " << *a << '\n';

    if (!b)
        std::cout << "error = " << b.error() << '\n';
}
```

This tiny program does more for your understanding than reading ten summaries.

46.9.2 Example: testing compiler mode support

```
cl /std:c++23 /EHsc main.cpp
```

```
clang++ -std=c++23 main.cpp -o app.exe
```

```
g++ -std=c++23 main.cpp -o app.exe
```

When a feature fails, do not guess immediately. Check:

- did the source code use the correct header,
- is the compiler in the correct language mode,
- is the local standard library new enough,
- does the compiler status page list the feature as implemented.

46.10 How professionals take notes from documentation

Reading documentation without taking notes is often inefficient.
A good professional note-taking pattern includes:

- feature name,
- required header or module,
- minimum standard,
- compiler version notes,
- key rules,
- common mistakes,
- minimal working example,
- one real project use case.

For example:

```
Feature: std::span
Header: <span>
Standard: C++20
Meaning: non-owning view over contiguous storage
Key rule: underlying storage must outlive the span
Use case: function parameters for arrays, vector, std::array
Common mistake: returning a span to destroyed data
```

This kind of note becomes a personal professional reference.

46.11 How to recognize outdated or weak information

Not every explanation on the internet is wrong, but a professional must know how to detect weakness.
Warning signs include:

- no version information,
- no compiler information,
- no build commands,
- examples that fail on modern compilers,
- vague phrases like “always” or “never” without context,
- claims that conflict with official vendor docs,
- articles that discuss modern features but show old usage patterns only.

When in doubt, go back to the primary source.

46.12 The most common mistakes beginners make with documentation

46.12.1 Reading only examples

Examples help, but examples are not the whole documentation.

46.12.2 Ignoring version support

A feature may be standardized but not yet available in the local toolchain.

46.12.3 Confusing language and library support

Sometimes the parser supports syntax, but the standard library implementation is still incomplete.

46.12.4 Ignoring notes and remarks

This often leads directly to wasted debugging time.

46.12.5 Reading secondary summaries instead of primary sources

Secondary summaries are useful for learning, but not for final verification.

46.12.6 Trying to memorize instead of understanding structure

A professional does not memorize every page. A professional learns where to find the truth quickly.

46.13 Building a personal documentation workflow

A strong long-term workflow might look like this:

1. start from the official language or tool source,
2. verify support status,
3. create a minimal test file,
4. compile in the correct standard mode,
5. record what worked,
6. keep a personal notebook of verified patterns.

Over time, this becomes one of your greatest strengths.

46.14 A practical reading routine for professionals

When faced with a new feature, library, or tool, use this routine:

46.14.1 Step 1: define the exact question

Bad question:

```
How do modules work?
```

Better question:

```
How do I compile a simple named module with my compiler on Windows?
```

46.14.2 Step 2: identify the correct official source

Is this a language question, compiler question, standard library question, or build-system question?

46.14.3 Step 3: read the signature or command syntax carefully

Do not skim the syntax.

46.14.4 Step 4: read notes and support details

This is where many hidden requirements appear.

46.14.5 Step 5: make a tiny experiment

A ten-line test is often enough.

46.14.6 Step 6: generalize only after verification

Do not move directly from one example to a whole architecture.

46.15 Windows-oriented professional practice

Since many developers work on Windows, it is important to keep a clean Windows-oriented documentation workflow.

46.15.1 MSVC workflow

When using MSVC:

- verify the standard mode,
- verify the installed Visual Studio toolset,
- read the Microsoft documentation for the feature,
- confirm whether the STL implementation supports the required library facility,
- compile a minimal test.

Typical commands:

```
cl /std:c++20 /EHsc main.cpp
```

```
cl /std:c++23 /EHsc main.cpp
```

```
cl /std:c++latest /EHsc main.cpp
```

46.15.2 Clang workflow on Windows

When using Clang on Windows, verify both the compiler and the library environment you are actually using.

```
clang++ -std=c++20 main.cpp -o app.exe
```

```
clang++ -std=c++23 main.cpp -o app.exe
```

```
clang++ -std=c++2c main.cpp -o app.exe
```

46.15.3 GCC workflow on Windows

With GCC-based environments on Windows:

```
g++ -std=c++20 main.cpp -o app.exe
```

```
g++ -std=c++23 main.cpp -o app.exe
```

```
g++ -std=c++2c main.cpp -o app.exe
```

In every case, documentation reading should be tied to the exact toolchain you actually run.

46.16 From reader to professional engineer

Reading official documentation is not only about fixing errors. It changes how you think.
You begin to:

- ask sharper questions,
- separate specification from opinion,
- verify assumptions before designing systems,
- understand why APIs are shaped the way they are,
- build software with fewer accidental mistakes,
- learn new tools faster.

This is one of the most powerful professional habits in programming.
A developer who reads official docs well can often learn new libraries, compilers, and frameworks far faster than someone who depends entirely on tutorials.

46.17 Professional advice for lifelong growth

A practical long-term habit is:

- every week, choose one official page and study it deeply,
- reproduce one example,
- add one note to your personal knowledge base,
- compare behavior across at least two compilers when relevant,
- revisit feature-status pages when you move to newer toolchains.

Over months and years, this discipline builds genuine expertise.

Summary

Reading official documentation is one of the defining skills of a professional C++ programmer.
It teaches you to:

- distinguish primary sources from secondary explanations,
- verify exact syntax, semantics, and support status,
- understand the difference between language, library, compiler, and build-system issues,
- transform documentation into tested working knowledge,

- avoid outdated or incorrect assumptions,
- build a reliable long-term learning process.

A professional does not need to memorize every technical detail in C++. That is impossible.
What matters more is this:

- knowing what question to ask,
- knowing where the official answer lives,
- knowing how to read it correctly,
- knowing how to verify it in code.

That skill is one of the strongest foundations of serious software engineering in Modern C++.

Compiler Differences

47.1 Why compiler differences matter

One of the most important turning points in becoming a professional C++ programmer is realizing that writing valid C++ code is only part of the job. The other part is understanding how different compilers interpret, diagnose, optimize, and support that code in practice.
The C++ standard defines the language and the standard library at a specification level, but real programs are compiled by concrete toolchains such as:

- MSVC,
- GCC,
- Clang.

These toolchains aim to implement the same language, but they still differ in many practical areas:

- standard-mode flags,
- conformance defaults,
- language extensions,
- warning systems,
- optimizer behavior,
- modules support,
- ABI and binary compatibility,
- standard library implementation maturity,
- diagnostics and analysis tooling.

A beginner often assumes that if code compiles on one compiler, then it is correct everywhere. A professional knows that this assumption is dangerous.

47.2 The three major toolchain families

In modern C++ work, the most important compiler families are:

47.2.1 MSVC

MSVC is the Microsoft compiler toolchain used heavily on Windows. It integrates deeply with Visual Studio, the Microsoft STL implementation, Windows SDKs, MSBuild, and many Windows-native development workflows.
Typical command:

```
cl /std:c++20 /EHsc main.cpp
```

47.2.2 GCC

GCC is one of the longest-standing and most widely used open-source compiler toolchains. It is common on Linux, available on Windows through MinGW-like environments, and closely associated with `libstdc++`.
Typical command:

```
g++ -std=c++20 main.cpp -o app.exe
```

47.2.3 Clang

Clang is part of the LLVM ecosystem. It is widely used for its diagnostics, tooling integration, static analysis, and fast front-end behavior. On Windows, Clang may be used with different library and linker environments depending on setup.
Typical command:

```
clang++ -std=c++20 main.cpp -o app.exe
```

47.3 Same language, different defaults

A crucial professional lesson is that the compilers do not start from identical defaults.
Even when all three support the same standard revision, they may still differ in:

- which standard mode is enabled by default,
- which nonstandard extensions are accepted silently,
- how strict diagnostics are,
- which warnings are enabled by default,
- which standard library revision is actually present.

This means portability is not automatic. It must be engineered deliberately.

47.4 Standard mode differences

The first practical compiler difference is the command-line flag used to enable a language version.

47.4.1 MSVC standard mode

MSVC uses `/std:...`.

```
cl /std:c++17 /EHsc main.cpp
cl /std:c++20 /EHsc main.cpp
cl /std:c++23preview /EHsc main.cpp
cl /std:c++latest /EHsc main.cpp
```

47.4.2 GCC standard mode

GCC uses `-std=...` and distinguishes between strict standard modes and GNU-extension modes.

```
g++ -std=c++17 main.cpp -o app.exe
g++ -std=c++20 main.cpp -o app.exe
g++ -std=c++23 main.cpp -o app.exe
g++ -std=c++2c main.cpp -o app.exe
```

GNU-extension modes:

```
g++ -std=gnu++17 main.cpp -o app.exe
g++ -std=gnu++20 main.cpp -o app.exe
g++ -std=gnu++23 main.cpp -o app.exe
g++ -std=gnu++2c main.cpp -o app.exe
```

47.4.3 Clang standard mode

Clang also uses `-std=...`.

```
clang++ -std=c++17 main.cpp -o app.exe
clang++ -std=c++20 main.cpp -o app.exe
clang++ -std=c++23 main.cpp -o app.exe
clang++ -std=c++2c main.cpp -o app.exe
```

47.4.4 Why this matters

A feature may fail for reasons that have nothing to do with the code itself. Common causes include:

- the wrong standard mode,
- an older library implementation,
- preview-only support,
- an experimental feature not enabled by default.

For professional work, never assume that a compiler uses the intended standard mode unless you set it explicitly.

47.5 Strict standard mode versus extension mode

Another major difference is how compilers handle implementation-specific extensions.

47.5.1 MSVC and conformance mode

MSVC historically supported a variety of Microsoft-specific behaviors. Modern conformance work is controlled heavily through `/permissive-` and related conformance switches.

```
cl /std:c++20 /permissive- /EHsc main.cpp
```

This is an important professional setting because code that compiles under permissive extension behavior may fail on GCC or Clang.

47.5.2 GCC strict mode versus GNU mode

GCC distinguishes clearly between:

- `-std=c++20`
- `-std=gnu++20`

The GNU mode allows GNU extensions in addition to the standard language.

```
g++ -std=c++20 main.cpp -o app.exe
g++ -std=gnu++20 main.cpp -o app.exe
```

In portable code, using strict standard mode is often safer.

47.5.3 Clang and extensions

Clang is also willing to support extensions, and its behavior may differ depending on compatibility goals and target environment. Clang documents many supported extensions explicitly.

47.5.4 Professional rule

When testing portability:

- prefer strict standard mode,
- avoid relying on vendor-only syntax,
- verify behavior under at least two compilers.

47.6 A concrete conformance example

Dependent names are a classic place where compiler differences and conformance settings matter.

```
template<typename T>
struct Wrapper
{
    template<typename U>
    static int value()
    {
```

```
        return 42;
    }
};

template<typename T>
int f()
{
    return Wrapper<T>::template value<int>();
}
```

In standard-conforming code, the `template` keyword after a dependent nested name is required. Code that omits it may still compile in more permissive settings on some toolchains or older modes, but fail in stricter conforming modes.
This is exactly the kind of issue that professionals must recognize: a compiler difference often reveals not arbitrary behavior, but whether code was depending on a nonconforming extension or a relaxed diagnostic mode.

47.7 Diagnostics quality differences

Compilers do not only differ in whether they accept code. They also differ in how they explain problems.

47.7.1 Clang diagnostics

Clang is widely respected for highly readable diagnostics. In many template-heavy and syntax-heavy cases, Clang often presents errors in a form that is easier to understand quickly.

47.7.2 GCC diagnostics

GCC diagnostics are strong and detailed, and in many areas they have improved significantly over time. GCC also provides powerful warning controls and detailed optimization-related options.

47.7.3 MSVC diagnostics

MSVC diagnostics are often deeply integrated into the Windows and Visual Studio workflow, including IDE navigation, code analysis, and project-based builds.

47.7.4 Professional consequence

When a difficult template or concepts error appears, it is often useful to test the same code under a second compiler. The second compiler may explain the problem more clearly.
This is not a sign of weakness. It is a professional debugging strategy.

47.8 Warning system differences

Warnings are one of the most important practical differences between compilers.

47.8.1 MSVC warnings

MSVC commonly uses warning levels such as:

```
cl /std:c++20 /EHsc /W4 main.cpp
cl /std:c++20 /EHsc /Wall main.cpp
```

47.8.2 GCC warnings

GCC warning workflows usually begin with:

```
g++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

Additional warning groups can be enabled as needed.

47.8.3 Clang warnings

Clang commonly uses a similar style:

```
clang++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

47.8.4 Important lesson

Similar flag names do not always mean identical warning behavior. For example:

- the exact warnings included in a warning group may differ,
- the wording of diagnostics differs,
- one compiler may warn where another stays silent,
- one compiler may treat a construct as an extension warning while another accepts it quietly.

47.8.5 Professional practice

A good cross-compiler warning policy is:

- enable high warning levels,
- treat warnings seriously,
- test under at least GCC or Clang in addition to MSVC when portability matters,
- avoid weakening warnings simply to silence correct diagnostics.

47.9 Standard library implementation differences

Even when the language front-end is similar, the standard library implementation may still differ.

47.9.1 Three common library environments

In practice, you may encounter:

- MSVC with Microsoft STL,
- GCC with `libstdc++`,
- Clang with either `libstdc++`, `libc++`, or Microsoft STL depending on platform and setup.

47.9.2 Why this matters

A language feature and a library feature are not the same thing.
Examples:

- the compiler may parse a modern feature successfully,
- but the associated standard library header may still be incomplete or unavailable in the active library implementation,
- or a newer library feature may be present under one library but not another.

47.9.3 Typical examples

This practical difference often appears when using newer library facilities such as:

- `std::print`,
- `std::expected`,
- `std::mdspan`,
- newer `<format>` improvements,
- newer ranges-related library components.

Code may compile under one toolchain and fail under another not because the language differs, but because the active standard library implementation has different maturity.

47.10 A language-support versus library-support example

Consider this example:

```
#include <expected>
#include <string>

std::expected<int, std::string> parse_positive(int x)
{
    if (x > 0)
        return x;

    return std::unexpected("value must be positive");
```

```
}

int main()
{
    auto r = parse_positive(-1);
    return r ? *r : 0;
}
```

If this fails, the reason may be one of several things:

- the standard mode is too old,
- the compiler front-end is too old,
- the standard library implementation is too old,
- the environment uses a different standard library than expected.

A professional never jumps to the conclusion that “the feature is broken.” The professional first identifies which layer is missing support.

47.11 ABI and binary compatibility differences

One of the deepest practical compiler differences is ABI.
ABI refers to low-level binary interface rules such as:

- name mangling,
- object layout assumptions,
- calling conventions,
- exception-handling model integration,
- runtime library compatibility.

47.11.1 Why ABI matters

If you compile one library with one compiler and another component with another compiler, binary compatibility may fail even if the source code is valid C++.
This is especially important for:

- dynamic libraries,
- plugin systems,
- mixed toolchain builds,
- distribution of prebuilt binary packages.

47.11.2 Professional consequence

Cross-compiler portability at the source level does not automatically imply cross-compiler binary compatibility.
That is why many professional systems standardize on:

- one compiler family,
- one standard library,
- one runtime configuration,
- one build configuration policy.

47.11.3 Source portability versus binary portability

These are different goals:

- **Source portability**: the same code can be recompiled under multiple compilers.
- **Binary portability**: the produced binaries interoperate safely without recompilation.

Professionals distinguish these carefully.

47.12 Object format and platform environment differences

Compiler differences are also shaped by platform environment.
On Windows, common differences may include:

- linker behavior,
- default runtime selection,
- import-library handling,
- debugging format,
- integration with Visual Studio or alternative toolchains.

On Linux or other systems, the same compiler family may operate in a very different runtime and linker environment.
This means portability is not only about compiler front ends. It is also about the surrounding toolchain ecosystem.

47.13 Modules support differences

Modules are a major example of practical compiler divergence.

47.13.1 The standard feature is one thing

C++20 standardizes modules as a language feature.

47.13.2 Real-world implementation is another thing

In practice, module workflows still differ among toolchains in areas such as:

- command-line structure,
- build-order requirements,
- generated artifacts,
- header unit handling,
- completeness of support,
- build-system integration.

47.13.3 GCC modules note

GCC documents module support and also documents that support is not complete in all areas.

47.13.4 Clang modules note

Clang distinguishes clearly between standard C++ modules and older Clang module mechanisms. Those are not the same model.

47.13.5 MSVC modules note

MSVC provides named-modules workflows integrated into its compiler and project environment, but command-line details and project integration still differ from GCC and Clang practice.

47.13.6 Professional conclusion

Do not assume that a module build recipe from one compiler transfers directly to another compiler unchanged.

47.14 A modules example

A minimal named-module example may look like this.

47.14.1 Module interface

```
// math.ixx
export module math;

export int add(int a, int b)
{
    return a + b;
}
```

47.14.2 Importer

```
// main.cpp
import math;
#include <iostream>

int main()
{
    std::cout << add(2, 3) << '\n';
}
```

The source code is simple, but the compilation workflow differs more across compilers than a traditional header-based build does.
That is exactly why compiler documentation matters.

47.15 Extensions and vendor-specific behavior

Each compiler family has its own extension history.

47.15.1 MSVC extensions

MSVC documents Microsoft-specific language extensions and the interaction with conformance switches.

47.15.2 GCC extensions

GCC has long supported GNU-specific behavior, especially in GNU dialect modes.

47.15.3 Clang extensions

Clang documents supported language extensions and feature-detection facilities.

47.15.4 Professional advice

Avoid vendor-specific extensions in portable code unless:

- the project is intentionally platform-specific,
- the extension is isolated behind a compatibility layer,
- the project documents the dependency clearly.

47.16 Feature-test macro differences

Modern compilers support feature-test macros to help code detect whether a language or library feature is available.
These are commonly macros such as:

```
#ifdef __cpp_concepts
#endif

#ifdef __cpp_lib_expected
#endif
```

Using these macros is often much more reliable than guessing from compiler brand alone.
For example, this is better:

```
#if defined(__cpp_lib_expected) && __cpp_lib_expected >= 202202L
    // use std::expected
#else
    // fallback
#endif
```

than this:

```
#ifdef _MSC_VER
    // assume feature exists
#endif
```

The first tests for the feature itself. The second tests only one vendor identity, which is weaker and often incorrect.

47.17 Optimization differences

Even when code is valid and portable, generated performance may differ substantially.
Compilers differ in:

- inlining heuristics,
- vectorization choices,
- constant-propagation aggressiveness,
- code layout strategies,
- debug-versus-release tradeoffs.

47.17.1 Professional implication

Never assume that performance behavior is identical across compilers.
If performance matters:

- benchmark under the real target compiler,
- benchmark under release settings,
- check generated code when appropriate,
- avoid myths based on a single toolchain.

47.18 Debugging and analysis differences

Compilers also differ in surrounding analysis and debugging support.
Examples include:

- warning granularity,
- static analysis integration,
- sanitizer availability,
- IDE integration,
- code browsing and refactoring tools.

A professional toolchain choice is often influenced not only by raw compilation success, but also by:

- diagnostics quality,
- debugging experience,
- build system compatibility,
- platform integration,
- library ecosystem maturity.

47.19 A portability example: newline and command-line assumptions

Even simple programs can accidentally become compiler- or environment-sensitive when they rely on weak assumptions.

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

int main()
{
    int n{};
    std::string line;

    std::cin >> n;
    std::getline(std::cin, line);

    std::cout << "line = [" << line << "]\n";
}
```

This example reveals an input-buffer issue that is not really a compiler difference, but it teaches an important lesson: many portability problems blamed on compilers are actually code-quality or environment-assumption issues.
A professional first separates:

- standard-defined behavior,

- implementation-defined behavior,
- unspecified behavior,
- undefined behavior,
- ordinary logic mistakes.

47.20 Undefined behavior and apparent compiler differences

A very important professional rule is this:
If code has undefined behavior, compiler differences are expected.
Example:

```
#include <iostream>

int main()
{
    int x = 1;
    int y = ++x + x++;
    std::cout << y << '\n';
}
```

If different compilers produce different results, that does not mean one compiler is wrong. It often means the program relied on behavior the language does not define safely.
Professionals do not treat undefined behavior as a compiler bug unless there is evidence of an actual conformance defect.

47.21 Practical cross-compiler workflow

A strong professional workflow for portable C++ is:

1. write code in a strict standard mode,
2. enable strong warnings,
3. test under at least two compilers,
4. avoid vendor-only extensions unless justified,
5. verify newer library features by feature-test macros when needed,
6. keep ABI boundaries clean,
7. isolate platform-specific code,
8. use CI or repeated local builds across toolchains.

47.22 A practical warning profile

A strong Windows-friendly baseline can look like this.

47.22.1 MSVC

```
cl /std:c++20 /permissive- /EHsc /W4 main.cpp
```

47.22.2 GCC

```
g++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

47.22.3 Clang

```
clang++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

This does not make code perfect, but it exposes many avoidable portability problems early.

47.23 A practical portability example with feature detection

```
#include <iostream>

#if defined(__has_include)
    #if __has_include(<expected>)
        #include <expected>
    #endif
#endif

int main()
{
#if defined(__cpp_lib_expected)
    std::cout << "std::expected is available\n";
#else
    std::cout << "std::expected is not available here\n";
#endif
}
```

This style is more professional than assuming availability based on compiler brand alone.

47.24 How to design code that survives compiler differences

47.24.1 Prefer standard language and library features

The more your code depends on standard facilities, the easier portability becomes.

47.24.2 Minimize compiler-specific conditionals

Avoid scattering vendor checks everywhere.
Prefer centralized compatibility headers.

47.24.3 Use narrow portability layers

For example, isolate:

- platform-specific file APIs,
- compiler-specific attributes,
- intrinsic usage,
- special calling conventions,
- OS-specific synchronization primitives.

47.24.4 Use build-system abstraction carefully

CMake and similar tools help reduce compiler-specific duplication, but the programmer still needs to understand what each compiler expects.

47.25 Common beginner mistakes

47.25.1 Assuming one successful build proves correctness

It does not.

47.25.2 Assuming every failure is a compiler bug

Often the cause is:

- wrong standard mode,
- missing header,
- incomplete library support,
- undefined behavior,
- extension-dependent code.

47.25.3 Treating compiler identity as feature identity

Compiler brand alone is not enough. Version, standard mode, and active standard library also matter.

47.25.4 Ignoring ABI boundaries

This can create subtle runtime failures even when compilation succeeds.

47.25.5 Turning off warnings too quickly

Warnings are often the first sign of nonportable assumptions.

47.26 A professional checklist

Before saying that code is portable, check:

- does it compile under the intended standard mode,
- does it avoid vendor-specific extensions,
- does it pass strong warning settings,
- does it avoid undefined behavior,
- does it use a supported standard library feature set,
- does it keep binary boundaries controlled,
- has it been tested under more than one toolchain.

47.27 Windows-oriented practical commands

47.27.1 MSVC strict build

```
cl /std:c++20 /permissive- /EHsc /W4 main.cpp
```

47.27.2 MSVC preview-style build

```
cl /std:c++latest /EHsc /W4 main.cpp
```

47.27.3 GCC strict build

```
g++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

47.27.4 GCC GNU-extensions build

```
g++ -std=gnu++20 -Wall -Wextra main.cpp -o app.exe
```

47.27.5 Clang strict build

```
clang++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

47.27.6 Clang post-C++23 experimentation

```
clang++ -std=c++2c -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

47.28 From compiler user to professional engineer

Beginners often ask:

Which compiler is best?

Professionals ask better questions:

- which compiler matches my deployment platform,
- which compiler gives the diagnostics I need,
- which standard library implementation is active,
- what ABI constraints apply,
- what feature set is mature enough for this project,
- which portability risks matter here.

This is the right mindset.

Summary

Compiler differences are not a side topic in C++. They are one of the central realities of professional work.
MSVC, GCC, and Clang all implement modern C++, but they differ in:

- standard-mode flags,
- extension defaults,
- warning systems,
- diagnostics,
- standard library environments,
- modules workflows,
- ABI behavior,
- optimization and tooling ecosystems.

A professional C++ programmer learns to work with these differences deliberately.
That means:

- writing standard-conforming code,
- enabling strong warnings,
- understanding feature support versus compiler identity,

- testing under multiple toolchains,
- respecting ABI boundaries,
- using compiler documentation as a primary source.

When you reach that stage, you are no longer merely writing C++ that happens to compile.
You are engineering C++ that is robust, portable, diagnosable, and professional.

Next Steps

48.1 Reaching the end of the first full journey

Finishing a complete book on Modern C++ is a major achievement. It means you have already crossed an important boundary:

- from isolated syntax to structured understanding,
- from reading examples to building programs,
- from basic statements to real design decisions,
- from beginner confusion to deliberate engineering practice.

But this point is not the end of the road. It is the beginning of a more serious phase. In professional C++, progress does not come from memorizing more syntax alone. It comes from building, testing, reading documentation, studying toolchains, improving design judgment, and developing consistency.
The next steps after this book should therefore be practical, structured, and realistic.

48.2 The correct mindset for the next stage

Many learners ask what they should study next, but the better question is this:

What kind of programmer should I become next?

That question matters because professional growth in C++ is not only about collecting topics. It is about developing capabilities such as:

- reading official documentation without fear,
- writing code that survives beyond one experiment,
- debugging systematically,
- understanding compiler behavior,
- using build systems correctly,
- designing reusable components,
- testing assumptions before trusting them.

The next stage is therefore not only **more C++**. It is **better engineering with C++**.

48.3 Strengthen the foundation through repetition

After a large technical book, many learners immediately jump into advanced topics. That is sometimes useful, but the first professional step is usually reinforcement.
Revisit and strengthen the most important parts of the foundation:

- value semantics,
- references and const-correctness,
- classes and invariants,
- RAII,
- smart pointers,
- containers and algorithms,
- error handling,
- templates and generic programming,
- concurrency basics,
- compile-time programming,
- modules and modern project structure.

The purpose of revision is not to reread everything passively. The purpose is to convert knowledge into reflexes.
A feature becomes professional knowledge only when you can use it correctly without confusion inside a real project.

48.4 Build more projects immediately

One of the biggest mistakes after finishing a major study phase is returning to passive learning only. Real progress now must come from project work.
You should build projects that force you to combine multiple parts of C++ together.
Good next-project categories include:

- command-line utilities,
- parser-based tools,
- configuration systems,
- file-processing tools,
- data transformation tools,
- simple HTTP clients or servers,
- small game or simulation utilities,

- reusable libraries,
- multi-module applications,
- tools that expose cross-platform behavior.

A project does not need to be huge. It needs to be real enough that design decisions matter.

48.4.1 Examples of strong next projects

A good next step is to create one project from each of the following groups.

48.4.2 Console productivity tools

Examples:

- note manager,
- code snippet organizer,
- TODO planner with priorities,
- CSV processor,
- directory scanner,
- backup utility.

48.4.3 Developer-oriented tools

Examples:

- expression parser,
- source-file statistics analyzer,
- simple formatter,
- mini configuration parser,
- dependency scanner,
- log analyzer.

48.4.4 Reusable libraries

Examples:

- string utility library,
- filesystem helper library,

- lightweight serialization layer,
- command-line parsing library,
- structured logging utility,
- validation framework.

These projects deepen your understanding far more than reading another fifty pages without implementation.

48.5 Move from single-file code to project structure

At the beginner stage, writing everything in one source file is acceptable. At the professional stage, it is not enough.
Your next goal should be to structure projects with:

- headers and source files,
- clear ownership of responsibilities,
- small interfaces,
- reusable helper functions,
- testable components,
- build-system support.

A useful progression is:

1. single-file prototype,
2. separated headers and source files,
3. static or reusable library,
4. tested project,
5. build-system-managed project,
6. multi-target project.

That progression teaches you how real software grows.

48.6 Learn CMake seriously

At this stage, one of the most important practical next steps is learning CMake properly.
A professional C++ developer is expected to understand more than raw compiler commands. You should know how to describe projects, targets, libraries, include paths, and dependencies in a maintainable way.
Your next CMake goals should include:

- creating a project with `project()`,
- defining executables with `add_executable()`,
- defining libraries with `add_library()`,
- linking with `target_link_libraries()`,
- setting include paths correctly,
- separating private and public usage requirements,
- using multiple source files and subdirectories,
- creating debug and release builds,
- understanding install rules and package-ready design.

48.6.1 A simple starting example

```
cmake_minimum_required(VERSION 3.20)
project(NextStepsApp LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

add_executable(next_steps
    src/main.cpp
    src/app.cpp
    src/app.hpp
)
```

This is simple, but it already introduces:

- project declaration,
- language selection,
- explicit C++ standard selection,
- source organization.

48.6.2 A library plus executable design

```
cmake_minimum_required(VERSION 3.20)
project(TaskSuite LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

add_library(task_core
```

```
    src/task.cpp
    src/task_manager.cpp
)

target_include_directories(task_core
    PUBLIC
        ${CMAKE_CURRENT_SOURCE_DIR}/include
)

add_executable(task_app
    src/main.cpp
)

target_link_libraries(task_app
    PRIVATE
        task_core
)
```

That kind of project layout is a strong professional next step.

48.7 Become comfortable with official documentation

One major difference between a learner and a professional is this:

> A learner asks whether a feature exists. A professional verifies it in the actual documentation and toolchain.

Your next step should be to develop a habit of checking official sources whenever you work with:

- a new standard library component,
- a compiler flag,
- a warning option,
- a feature that depends on C++20, C++23, or later,
- modules support,
- build-system behavior,
- standard library implementation maturity.

You should become comfortable reading:

- standard library reference pages,
- compiler status pages,
- warning and optimization documentation,
- build-system manuals,
- tool-specific documentation.

This habit protects you from outdated examples and false assumptions.

48.8 Practice across compilers

A very strong next step is to stop trusting only one toolchain.
Even if Windows is your main environment, you should test your projects under more than one compiler when possible. This improves code quality because it reveals:

- hidden assumptions,
- extension-dependent code,
- warning differences,
- incomplete portability,
- standard-mode mistakes,
- library support misunderstandings.

48.8.1 A good baseline

On Windows, a strong baseline is:

- MSVC in a strict standard mode,
- Clang in standard mode,
- GCC if available in your environment.

Typical commands:

48.8.2 MSVC

```
cl /std:c++20 /permissive- /EHsc /W4 main.cpp
```

48.8.3 Clang

```
clang++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

48.8.4 GCC

```
g++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

Once this becomes normal, your code quality will improve quickly.

48.9 Learn debugging as a first-class skill

At the next professional stage, debugging should become a formal skill rather than something done randomly.
You should train yourself to debug in a disciplined way:

- reproduce the problem,
- minimize the case,
- isolate the failing component,
- verify inputs,
- inspect assumptions,
- reduce the code to the smallest failing example,
- compare behavior under another compiler if needed.

48.9.1 A small debugging example

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

int main()
{
    std::vector<int> values{1, 2, 3};

    for (std::size_t i = 0; i <= values.size(); ++i)
        std::cout << values[i] << '\n';
}
```

The problem here is not mysterious. It is a bounds mistake. But the professional lesson is important:

- isolate the logic,
- identify the exact invalid condition,
- fix the rule,
- test again.

The corrected loop is:

```
for (std::size_t i = 0; i < values.size(); ++i)
    std::cout << values[i] << '\n';
```

48.10 Invest in testing

A professional next step is to stop thinking of testing as optional.
For every useful project you build, test at least these categories:

- normal valid input,
- boundary values,
- invalid input,
- empty data,
- large input,
- repeated operations,
- save/load cycles if persistence exists.

48.10.1 Example of simple assertion-based testing

```
#include <cassert>
#include <string>

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

int main()
{
    assert(add(2, 3) == 5);
    assert(add(-1, 1) == 0);
    assert(add(0, 0) == 0);
}
```

Even simple testing improves reliability.
Your next step after this level can include a real unit-testing framework, but the important thing now is to build the habit.

48.11 Study the standard library more deeply

At this point, the standard library should become one of your central areas of continued growth.
Important next-study areas include:

- algorithms,
- ranges,
- containers in depth,
- strings and text handling,

- formatting,
- filesystem,
- chrono,
- thread support,
- atomics,
- optional and expected,
- span and mdspan,
- memory facilities.

Do not study them as isolated names. Study them through real problems.
For example:

- use `std::span` when designing safer buffer interfaces,
- use `std::expected` in explicit error-return designs,
- use `std::filesystem` in real file tools,
- use `std::format` or `std::print` in clear output code,
- use ranges for readable data pipelines.

48.12 Deepen generic programming gradually

Templates, concepts, and generic design should continue to be part of your next steps, but not in a purely abstract way.
A good next progression is:

1. write ordinary function templates,
2. write class templates,
3. use standard traits,
4. constrain templates with concepts,
5. design reusable generic utilities,
6. study compile-time behavior and diagnostics,
7. test generic code under multiple compilers.

48.12.1 A small concept-based example

```
#include <concepts>
#include <iostream>

template<typename T>
concept Number = std::integral<T> || std::floating_point<T>;

template<Number T>
T twice(T value)
{
    return value + value;
}

int main()
{
    std::cout << twice(10) << '\n';
    std::cout << twice(2.5) << '\n';
}
```

The point is not only that this works. The point is that your interfaces become clearer and misuse becomes harder.

48.13 Learn design, not only syntax

Professional growth eventually depends more on design quality than on remembering keywords.
Your next design goals should include:

- maintaining class invariants,
- preferring clear ownership,
- minimizing global state,
- separating interface from implementation,
- reducing coupling,
- keeping components small and focused,
- making invalid states harder to represent,
- choosing composition when it is simpler and safer.

48.13.1 A simple design comparison

Weak design:

```
struct User
{
    std::string name;
    int age;
};
```

This allows invalid values too easily.
A stronger design may validate construction:

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

class User
{
public:
    User(std::string name, int age)
        : name_(std::move(name)), age_(age)
    {
        if (name_.empty())
            throw std::invalid_argument("name must not be empty");

        if (age_ < 0)
            throw std::invalid_argument("age must not be negative");
    }

    const std::string& name() const noexcept { return name_; }
    int age() const noexcept { return age_; }

private:
    std::string name_;
    int age_;
};
```

That is a small example of the professional shift from raw data to protected invariants.

48.14 Study performance with discipline

Performance is an important next step, but it should be approached professionally.
This means:

- do not guess performance,
- do not optimize randomly,
- measure first,
- understand algorithmic complexity,
- inspect copies and allocations,
- use the standard library wisely,
- compare release builds, not only debug builds.

Good next performance topics include:

- value categories and move semantics,

- avoiding unnecessary allocations,
- choosing the right container,
- algorithm complexity,
- memory access patterns,
- string and formatting cost,
- concurrency overhead,
- profiling methodology.

48.15 Explore specialization paths

After a strong general foundation, the next stage often includes choosing one or more specialization directions.
Possible directions include:

- systems programming,
- backend and networking,
- GUI application development,
- game and simulation programming,
- scientific or numerical computing,
- compilers and language tools,
- embedded programming,
- high-performance computing,
- security-focused tooling,
- cross-platform desktop tools.

The correct path depends on your goals, but the important point is this: specialization should grow on top of a strong general foundation, not replace it too early.

48.16 Create a personal portfolio of serious work

The next step to professionalism is not only learning more. It is showing the results.
A strong portfolio should include projects that demonstrate:

- clean structure,
- modern standard usage,

- real input and output,
- tested behavior,
- readable design,
- good build instructions,
- cross-compiler awareness,
- meaningful documentation.

Useful portfolio project ideas include:

- a parser,
- a file analyzer,
- a reusable library,
- a configuration engine,
- a task planner,
- a small backend service,
- a modular desktop tool,
- a domain-specific utility.

48.17 Write and explain what you learn

One of the strongest next steps is teaching, even at a small scale.
When you write notes, articles, or internal documentation, you are forced to clarify:

- what you actually understand,
- what is still vague,
- which examples are truly minimal,
- which design rules are genuinely useful.

A simple example is keeping a personal technical notebook.

48.17.1 Example note format

```
Topic: std::span
What it is: non-owning view over contiguous storage
Header: <span>
Main rule: underlying data must outlive the span
When to use: function parameters for arrays, vector, std::array
Common mistake: returning a span to temporary or destroyed storage
```

That practice builds long-term professional memory.

48.18 Develop a sustainable weekly routine

The next stage becomes much easier if you use a structured weekly rhythm.
A strong routine might include:

- one official documentation topic,
- one small code experiment,
- one improvement to an existing project,
- one debugging exercise,
- one build-system improvement,
- one review of warnings or compiler behavior,
- one written summary of what was learned.

This is more powerful than chaotic study because it compounds over time.

48.19 Use the language standard intentionally

As you continue, you should become more deliberate about standard selection.
A professional should know when a project is targeting:

- C++17,
- C++20,
- C++23,
- or preview-style newer modes.

The next step is not merely using the newest syntax. It is selecting the right standard for the real toolchain, team, deployment target, and library availability.

48.20 Continue learning C++20, C++23, and the direction of C++26

Modern C++ is still evolving. Your next steps should therefore include continued awareness of newer facilities such as:

- ranges,
- coroutines,
- modules,
- compile-time programming improvements,
- source-location tools,

- formatting and printing,
- expected-style error handling,
- multidimensional views,
- the emerging direction of reflection and contracts.

The key professional rule is:

> Study new features, but verify actual compiler and library support before relying on them in serious projects.

48.21 From learner to professional

The transition to professionalism is not one single moment. It happens gradually when your habits change.
You become more professional when you:

- verify instead of guessing,
- design instead of patching,
- test instead of hoping,
- structure instead of piling code together,
- measure instead of assuming,
- read documentation instead of relying only on summaries,
- build repeatedly instead of consuming passively.

That transformation matters more than any single advanced feature.

48.22 A practical 90-day plan

A useful way to convert this chapter into action is to follow a 90-day progression.

48.22.1 Days 1 to 30

- rebuild one project from this book from scratch,
- separate it into headers and source files,
- compile it under at least two compilers,
- add a CMake build,
- write basic tests,
- improve error handling.

48.22.2 Days 31 to 60

- build one new medium-sized console project,
- use filesystem, format or print, optional or expected where appropriate,
- add persistence or configuration support,
- reduce warnings to zero under strict settings,
- document the build process clearly.

48.22.3 Days 61 to 90

- build a reusable library or a multi-module application,
- use stronger design boundaries,
- write a short technical explanation of the architecture,
- test under multiple toolchains,
- review one official documentation topic each week.

That kind of plan creates real momentum.

48.23 Final advice

At this point, the best next step is not to chase every advanced topic at once.
Instead:

- build regularly,
- structure projects carefully,
- read official documentation,
- learn CMake,
- test under real toolchains,
- strengthen the standard library,
- improve design judgment,
- keep a written record of what you verify and learn.

This is how strong C++ developers are built.

Summary

This book aimed to take you from the beginning of Modern C++ to a serious working foundation. The next stage is now yours to build.
Your next professional steps should include:

- strengthening fundamentals through implementation,
- building more projects,
- moving to real project structure,
- learning CMake seriously,
- reading official documentation confidently,
- testing across compilers,
- improving debugging and testing discipline,
- studying design more deeply,
- creating a visible portfolio of real work,
- continuing with modern features carefully and intentionally.

The most important final lesson is simple:

> Do not stop at knowing C++. Continue until you can use C++ professionally, repeatedly, and with confidence.

That is the real next step.

Part XIV

Appendices and References

Appendices

Appendix A: Glossary

This glossary collects essential Modern C++ terms used throughout the book. It is designed as a fast professional reference rather than a historical dictionary.

ABI

Application Binary Interface. The low-level binary contract that affects how compiled code interacts across object files and libraries, including calling conventions, name mangling, object layout, exception model behavior, and runtime compatibility.

Algorithm

A reusable function from the standard library that operates on iterator ranges or ranges, such as `std::sort`, `std::find`, and `std::ranges::transform`.

Argument

A value passed to a function, constructor, or template.

Array

A fixed-size contiguous sequence of elements. In C++, this may mean a built-in array such as `int a[10];` or a safer standard wrapper such as `std::array`.

Attribute

A standardized or compiler-specific annotation written with double brackets such as `[[nodiscard]]`, `[[maybe_unused]]`, or `[[deprecated]]`.

Automatic storage duration

Objects whose lifetime is tied to a scope, typically local variables inside functions.

Binding

Associating a name with an entity such as an object, reference, function, or template parameter.

Class

A user-defined type that groups data and behavior. In C++, `class` and `struct` are nearly identical except for default access and inheritance rules.

Compilation unit

A source file after preprocessing, together with all included text. In normal practice, each `.cpp` file becomes one compilation unit.

Concept

A compile-time constraint on template parameters introduced in C++20. Concepts improve diagnostics and make generic interfaces clearer.

Const-correctness

The discipline of marking values, member functions, and interfaces as `const` whenever mutation is not intended.

Constant expression

An expression that can be evaluated during translation under the rules of constant evaluation.

Constructor

A special member function that initializes an object.

Copy semantics

Behavior based on creating a new object as a copy of another object.

Coroutine

A suspendable function introduced in C++20 that uses `co_await`, `co_yield`, or `co_return`.

Dangling reference

A reference that refers to an object that no longer exists.

Declaration

A statement that introduces a name and its type or form.

Definition

A declaration that fully specifies an entity or allocates storage when required.

Destructor

A special member function that runs when an object is destroyed. It is central to RAII.

Dynamic allocation

Allocation of storage at runtime, typically through facilities such as `new`, `std::make_unique`, or `std::make_shared`.

Encapsulation

The design principle of grouping data and operations together while controlling access to internal state.

Expression

A combination of values, operators, and function calls that produces a result.

Forward declaration

A declaration that introduces a type or function name before the full definition is provided.

Function overloading

Defining multiple functions with the same name but different parameter lists.

Generic programming

Programming with templates and abstractions that work across types while preserving correctness and efficiency.

Header

A file typically ending in `.h` or `.hpp` that contains declarations shared by multiple source files.

Header guard

A preprocessor technique used to prevent a header from being included multiple times in the same translation unit.

Inheritance

An object-oriented mechanism through which one class derives from another.

Invariant

A rule that must remain true for an object or system to be considered valid.

Iterator

An object used to traverse elements in a sequence.

Lambda

An unnamed function object written using lambda syntax such as `[](int x){ return x * 2; }`.

Lifetime

The period during which an object exists and may be used safely.

Lvalue

An expression that identifies a persistent object.

Memory leak

Allocated memory that is never released because ownership was lost.

Module

A language-level unit introduced in C++20 that provides a semantic import model as an alternative to textual inclusion by headers.

Move semantics

Transfer-oriented behavior that allows resources to be moved from one object to another instead of copied.

Name mangling

Compiler-generated encoding of symbol names used to represent function overloads, namespaces, templates, and other language details in object files.

Namespace

A scope used to organize names and avoid collisions.

Object

A region of storage with a type and lifetime.

Overload resolution

The compile-time process that selects the best matching overloaded function.

Ownership

The responsibility for managing the lifetime of a resource.

Polymorphism

A design in which a common interface supports multiple concrete behaviors. In C++, this can be dynamic via virtual functions or static via templates and CRTP-like techniques.

Preprocessor

The phase that processes directives such as `#include`, `#define`, and conditional compilation.

RAII

Resource Acquisition Is Initialization. A core C++ idiom where resource lifetime is tied to object lifetime.

Range

A sequence abstraction used by the standard library, especially in C++20 ranges.

Reference

An alias to an existing object.

Rvalue

An expression typically representing a temporary value or movable value.

Scope

The region of code where a name can be used.

Smart pointer

A library type that manages dynamic resources through ownership rules, such as `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`.

Static storage duration

Storage that exists for the lifetime of the program.

Template

A compile-time mechanism for parameterized code generation over types, values, or templates.

Thread

An execution path that can run concurrently with others.

Translation unit

A source file after preprocessing, ready for compilation.

Undefined behavior

Program behavior for which the C++ language imposes no requirements. Professional code should avoid it rigorously.

Value semantics

A style in which objects behave like values: they can be copied, moved, compared, and reasoned about independently.

View

A non-owning abstraction over a sequence or data source, such as `std::span` or many ranges views.

Appendix B: Common beginner mistakes

This appendix summarizes mistakes that repeatedly slow down new C++ programmers. The goal is not criticism. The goal is faster progress.

Mistake 1: Mixing declaration, ownership, and lifetime ideas

Many beginners can write pointers, references, objects, and smart pointers syntactically, but do not yet understand who owns what and how long it stays alive.
Weak example:

```
int* make_value()
{
    int x = 42;
    return &x;
}
```

The returned pointer dangles because `x` is destroyed when the function ends.
Better example:

```
int make_value()
{
    return 42;
}
```

Or, when dynamic lifetime is needed:

```
#include <memory>

std::unique_ptr<int> make_value()
{
    return std::make_unique<int>(42);
}
```

Mistake 2: Using raw `new` and `delete` too early

Beginners often learn dynamic allocation from very old examples and then overuse manual memory management.
Weak example:

```
int* p = new int(10);
// ...
delete p;
```

Better:

```
auto p = std::make_unique<int>(10);
```

Mistake 3: Writing giant functions

Large functions hide bugs, mix responsibilities, and make testing harder.
Weak example:

```
void process()
{
    // reading, validation, parsing, computation,
    // formatting, printing, saving, and cleanup
    // all mixed together
}
```

Better: divide the logic into small focused functions.

```
std::string read_input();
bool validate_input(const std::string& text);
int compute_value(const std::string& text);
void print_result(int value);
```

Mistake 4: Forgetting `const`

Beginners often leave everything mutable.
Weak example:

```
void print_name(std::string& name)
{
    std::cout << name << '\n';
}
```

Better:

```
void print_name(const std::string& name)
{
    std::cout << name << '\n';
}
```

Mistake 5: Using C-style arrays and strings when safer library types fit better

Weak example:

```
char name[100];
std::cin >> name;
```

Better:

```
std::string name;
std::getline(std::cin, name);
```

Mistake 6: Ignoring return values and errors

Weak example:

```
std::ifstream file("data.txt");
// assume success without checking
```

Better:

```
std::ifstream file("data.txt");
if (!file)
{
    std::cerr << "Failed to open file\n";
}
```

Mistake 7: Using `using namespace std;` everywhere

This seems convenient early on, but it scales poorly and can cause name collisions.

Weak example:

```
using namespace std;
```

Better: qualify names explicitly or use narrow local aliases when justified.

```
std::vector<int> values;
std::string name;
```

Mistake 8: Not understanding pass-by-value versus pass-by-reference

Weak example:

```
void print(const std::vector<int> values)
{
    for (int v : values)
        std::cout << v << ' ';
}
```

This copies the whole vector.

Better:

```
void print(const std::vector<int>& values)
{
    for (int v : values)
        std::cout << v << ' ';
}
```

Mistake 9: Reinventing the standard library

Beginners sometimes build weak substitutes for containers, strings, smart pointers, or algorithms before mastering the standard ones.
Professional advice: learn `std::vector`, `std::string`, algorithms, smart pointers, `std::optional`, `std::expected`, `std::span`, and ranges before creating custom replacements.

Mistake 10: Treating compiler success as proof of correctness

Code that compiles is not necessarily correct, safe, portable, or maintainable.

Mistake 11: Ignoring warnings

Warnings are often the first sign of weak assumptions.
Recommended habit on Windows:

```
cl /std:c++20 /permissive- /EHsc /W4 main.cpp
clang++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
g++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

Mistake 12: Using old C conversion style carelessly

Weak example:

```
double x = 3.7;
int y = (int)x;
```

Better:

```
double x = 3.7;
int y = static_cast<int>(x);
```

Mistake 13: Writing classes without invariants

Weak design:

```
struct BankAccount
{
    double balance;
};
```

Better:

```
#include <stdexcept>

class BankAccount
{
public:
    explicit BankAccount(double initial_balance)
        : balance_(initial_balance)
    {
```

```
        if (initial_balance < 0.0)
            throw std::invalid_argument("negative initial balance");
    }

    double balance() const noexcept
    {
        return balance_;
    }

    void deposit(double amount)
    {
        if (amount < 0.0)
            throw std::invalid_argument("negative deposit");
        balance_ += amount;
    }

private:
    double balance_{};
};
```

Mistake 14: Misunderstanding object slicing

Weak example:

```
#include <iostream>

struct Base
{
    virtual ~Base() = default;
    virtual void print() const
    {
        std::cout << "Base\n";
    }
};

struct Derived : Base
{
    void print() const override
    {
        std::cout << "Derived\n";
    }
};

void show(Base b)
{
    b.print();
}
```

Passing by value slices the object.

Better:

```
void show(const Base& b)
```

```
{
    b.print();
}
```

Mistake 15: Using inheritance where composition is simpler

Not every relationship should be modeled with a base class.

Mistake 16: Confusing `std::move` with movement itself

`std::move` is a cast to an rvalue expression. Real movement depends on the receiving operation.

Mistake 17: Returning views to dead storage

Weak example:

```
#include <span>
#include <vector>

std::span<int> bad_view()
{
    std::vector<int> v{1, 2, 3};
    return v;
}
```

Better: return an owning object, or ensure the viewed storage outlives the view.

Mistake 18: Overusing macros for ordinary tasks

Prefer `constexpr`, `inline`, templates, enums, and functions whenever possible.

Mistake 19: Not separating interface from implementation

Beginners often place everything in one file for too long. Professional growth requires headers, source files, libraries, or modules.

Mistake 20: Learning from outdated examples without checking current support

Modern C++ changes significantly across standards and implementations. Always verify support with the actual compiler and standard library you use.

Appendix C: Cheat sheets

This appendix is a compact quick-reference section for daily work.

Cheat sheet: Basic declarations

```
int x = 10;
double pi = 3.14159;
char ch = 'A';
bool ok = true;
std::string name = "Modern C++";
```

Cheat sheet: References and pointers

```
int value = 42;

int& ref = value;
const int& cref = value;

int* ptr = &value;
const int* cptr = &value;
```

Cheat sheet: `const`

```
const int x = 5;
const std::string name = "Ayman";

void print(const std::string& text);
```

Cheat sheet: Value categories in practice

```
std::string s = "abc";          // object
std::string t = s;              // copy from lvalue
std::string u = std::move(s);   // move from xvalue
```

Cheat sheet: Function forms

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

void print(const std::string& text)
{
    std::cout << text << '\n';
}

auto square(double x) -> double
{
    return x * x;
}
```

Cheat sheet: Class basics

```
class Point
```

```
{
public:
    Point(int x, int y) : x_(x), y_(y) {}

    int x() const noexcept { return x_; }
    int y() const noexcept { return y_; }

private:
    int x_;
    int y_;
};
```

Cheat sheet: RAII

```
#include <fstream>

void write_file()
{
    std::ofstream file("log.txt");
    if (file)
        file << "hello\n";
}
```

Cheat sheet: `std::vector`

```
std::vector<int> values{1, 2, 3};

values.push_back(4);
values.pop_back();

for (int v : values)
    std::cout << v << ' ';
```

Cheat sheet: `std::array`

```
std::array<int, 3> values{10, 20, 30};
```

Cheat sheet: `std::string`

```
std::string name = "Modern";
name += " C++";

std::cout << name.size() << '\n';
```

Cheat sheet: Range-based `for`

```
for (const auto& value : values)
{
    std::cout << value << '\n';
}
```

Cheat sheet: Algorithms

```
#include <algorithm>
#include <vector>

std::vector<int> v{5, 1, 4, 2, 3};

std::sort(v.begin(), v.end());

auto it = std::find(v.begin(), v.end(), 4);
```

Cheat sheet: Lambdas

```
auto twice = [](int x)
{
    return x * 2;
};

std::cout << twice(10) << '\n';
```

Cheat sheet: Smart pointers

```
#include <memory>

auto p1 = std::make_unique<int>(42);
auto p2 = std::make_shared<std::string>("text");
std::weak_ptr<std::string> weak = p2;
```

Cheat sheet: `std::optional`

```
#include <optional>

std::optional<int> parse(bool ok)
{
    if (ok)
        return 42;
    return std::nullopt;
}
```

Cheat sheet: `std::expected`

```
#include <expected>
#include <string>

std::expected<int, std::string> parse_positive(int x)
{
    if (x > 0)
        return x;

    return std::unexpected("value must be positive");
}
```

Cheat sheet: `std::span`

```
#include <span>
#include <vector>

void print_all(std::span<const int> values)
{
    for (int v : values)
        std::cout << v << ' ';
}

std::vector<int> data{1, 2, 3, 4};
print_all(data);
```

Cheat sheet: Ranges

```
#include <ranges>
#include <vector>

std::vector<int> data{1, 2, 3, 4, 5, 6};

auto result =
    data
    | std::views::filter([](int x) { return x % 2 == 0; })
    | std::views::transform([](int x) { return x * x; });
```

Cheat sheet: `constexpr`

```
constexpr int square(int x)
{
    return x * x;
}

constexpr int value = square(7);
```

Cheat sheet: `consteval`

```
consteval int checked(int x)
{
    return x > 0 ? x : throw "invalid";
}
```

Cheat sheet: `constinit`

```
constinit int global_counter = 0;
```

Cheat sheet: `std::source_location`

```
#include <source_location>
#include <string_view>
```

```
void log(
    std::string_view text,
    const std::source_location& loc = std::source_location::current())
{
    std::cout << loc.file_name() << ':' << loc.line() << ' ' << text << '\n';
}
```

Cheat sheet: `std::format`

```
#include <format>
#include <string>

std::string text = std::format("id = {}, value = {:.2f}", 7, 3.5);
```

Cheat sheet: `std::print`

```
#include <print>

int main()
{
    std::print("Hello, {}\n", "Modern C++");
}
```

Cheat sheet: Threads and mutexes

```
#include <mutex>
#include <thread>

std::mutex m;

void work()
{
    std::scoped_lock lock(m);
}
```

Cheat sheet: Filesystem

```
#include <filesystem>

std::filesystem::path p = "data.txt";

if (std::filesystem::exists(p))
{
    // ...
}
```

Cheat sheet: Modules

```
// math.ixx
export module math;
```

```cpp
export int add(int a, int b)
{
    return a + b;
}
```

```cpp
// main.cpp
import math;
```

Appendix D: Compiler commands

This appendix gives practical Windows-oriented command examples for common compilers. Exact support depends on the installed compiler version, standard library, and environment.

MSVC basic commands

```
cl /EHsc main.cpp
cl /std:c++17 /EHsc main.cpp
cl /std:c++20 /EHsc main.cpp
cl /std:c++latest /EHsc main.cpp
```

MSVC stronger conformance and warnings

```
cl /std:c++20 /permissive- /EHsc /W4 main.cpp
cl /std:c++latest /permissive- /EHsc /W4 main.cpp
```

MSVC multiple files

```
cl /std:c++20 /EHsc main.cpp app.cpp util.cpp
```

Clang basic commands on Windows

```
clang++ -std=c++17 main.cpp -o app.exe
clang++ -std=c++20 main.cpp -o app.exe
clang++ -std=c++23 main.cpp -o app.exe
clang++ -std=c++2c main.cpp -o app.exe
```

Clang with warnings

```
clang++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

GCC basic commands on Windows

```
g++ -std=c++17 main.cpp -o app.exe
g++ -std=c++20 main.cpp -o app.exe
g++ -std=c++23 main.cpp -o app.exe
g++ -std=c++2c main.cpp -o app.exe
```

GCC GNU dialect variants

```
g++ -std=gnu++20 main.cpp -o app.exe
g++ -std=gnu++23 main.cpp -o app.exe
```

GCC with warnings

```
g++ -std=c++20 -Wall -Wextra -Wpedantic main.cpp -o app.exe
```

Building with multiple files

```
g++ -std=c++20 main.cpp app.cpp util.cpp -o app.exe
clang++ -std=c++20 main.cpp app.cpp util.cpp -o app.exe
```

Compiling modules: conceptual examples

Real module workflows vary by toolchain and version. The following commands are representative examples and may need adaptation to the actual compiler release in use.

MSVC named module example

```
cl /std:c++20 /EHsc /c math.ixx
cl /std:c++20 /EHsc main.cpp math.obj
```

Clang named module example

```
clang++ -std=c++20 --precompile math.ixx -o math.pcm
clang++ -std=c++20 main.cpp math.pcm -o app.exe
```

GCC named module example

```
g++ -std=c++20 -fmodules -c math.ixx
g++ -std=c++20 -fmodules main.cpp math.o -o app.exe
```

CMake configure and build

```
cmake -S . -B build
cmake --build build
```

CMake release build

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
```

Useful professional habits

- Set the standard mode explicitly.
- Enable meaningful warning levels.

- Use at least two compilers for portability-sensitive code.
- Verify new library-feature support on the actual standard library implementation in use.

Appendix E: Modern replacements vs legacy techniques

This appendix summarizes common transitions from older styles to clearer Modern C++ practices.

Raw arrays to `std::array` or `std::vector`

Legacy:

```cpp
int data[5] = {1, 2, 3, 4, 5};
```

Modern:

```cpp
std::array<int, 5> data{1, 2, 3, 4, 5};
```

Or, when runtime size is needed:

```cpp
std::vector<int> data{1, 2, 3, 4, 5};
```

Pointer + length to `std::span`

Legacy:

```cpp
void print_values(const int* data, std::size_t count);
```

Modern:

```cpp
void print_values(std::span<const int> values);
```

Manual `new`/`delete` to smart pointers

Legacy:

```cpp
Widget* w = new Widget();
// ...
delete w;
```

Modern:

```cpp
auto w = std::make_unique<Widget>();
```

Null pointers as ownership markers to `std::optional` or smart pointers

Legacy:

```cpp
User* find_user(int id);
```

Modern option for non-owning result-with-presence meaning:

```cpp
std::optional<User> find_user(int id);
```

Error codes and out parameters to `std::expected`

Legacy:

```
bool parse_value(const std::string& text, int& result);
```

Modern:

```
std::expected<int, std::string> parse_value(const std::string& text);
```

Macros for constants to `constexpr`

Legacy:

```
#define BUFFER_SIZE 1024
```

Modern:

```
constexpr int buffer_size = 1024;
```

Macros for typed helpers to inline functions or templates

Legacy:

```
#define SQUARE(x) ((x) * (x))
```

Modern:

```
template<typename T>
constexpr T square(T x)
{
    return x * x;
}
```

C-style casts to named C++ casts

Legacy:

```
int x = (int)3.7;
```

Modern:

```
int x = static_cast<int>(3.7);
```

Character buffers to `std::string`

Legacy:

```
char name[128];
```

Modern:

```
std::string name;
```

Manual formatting with streams everywhere to `std::format`

Legacy:

```
std::ostringstream out;
out << "x = " << x << ", y = " << y;
```

Modern:

```
std::string text = std::format("x = {}, y = {}", x, y);
```

Verbose output chains to `std::print`

Legacy:

```
std::cout << "value = " << value << '\n';
```

Modern:

```
std::print("value = {}\n", value);
```

Manual file-path strings to `std::filesystem::path`

Legacy:

```
std::string path = "data\\file.txt";
```

Modern:

```
std::filesystem::path path = "data/file.txt";
```

Traditional iterator-heavy data pipelines to ranges

Legacy:

```
std::vector<int> out;

for (int v : data)
{
    if (v % 2 == 0)
        out.push_back(v * v);
}
```

Modern:

```
auto out =
    data
    | std::views::filter([](int v) { return v % 2 == 0; })
    | std::views::transform([](int v) { return v * v; });
```

Headers-only organization by habit to modules where appropriate

Legacy model:

```
#include "math.hpp"
```

Modern direction:

```
import math;
```

Handwritten source-location macros to `std::source_location`

Legacy:

```
#define LOG(msg) log_impl(msg, __FILE__, __LINE__)
```

Modern:

```
void log(
    std::string_view msg,
    const std::source_location& loc = std::source_location::current());
```

Manual multidimensional indexing to `std::mdspan`

Legacy:

```
int value = buffer[row * cols + col];
```

Modern:

```
std::mdspan<int, std::dextents<std::size_t, 2>> m(buffer.data(), rows, cols);
int value = m[row, col];
```

Guiding rule

Modern replacement does not mean replacing old syntax blindly. It means choosing safer, clearer, more expressive standard tools when they fit the design.

Appendix F: Feature evolution by standard

This appendix groups major language and library directions by standard revision. It is intentionally practical rather than exhaustive.

Before C++11

Typical older style included:

- manual memory management with raw `new`/`delete`,
- no move semantics,

- no standard smart pointers in modern form,
- no `auto` type deduction in modern meaning,
- no lambdas,
- no range-based `for`,
- no `constexpr`,
- no `std::thread`,
- no `std::unique_ptr`,
- no `std::optional`,
- no ranges, modules, concepts, or coroutines.

C++11

This revision changed modern C++ permanently.
Major additions included:

- move semantics and rvalue references,
- `auto`,
- lambdas,
- range-based `for`,
- `nullptr`,
- `enum class`,
- smart pointers such as `std::unique_ptr` and `std::shared_ptr`,
- `std::thread`, `std::mutex`, atomics, and condition variables,
- `constexpr` in early form,
- variadic templates,
- uniform initialization,
- `std::array`, `std::tuple`, `std::chrono`,
- stronger type-traits and generic-programming facilities.

C++14

This revision refined C++11 and improved usability.
Major additions included:

- generic lambdas,
- return-type deduction for normal functions,
- improved `constexpr`,
- `std::make_unique`,
- additional library refinements and fixes.

C++17

This revision made Modern C++ far more practical for everyday development.
Major additions included:

- structured bindings,
- `if constexpr`,
- fold expressions,
- inline variables,
- class template argument deduction,
- `std::optional`,
- `std::variant`,
- `std::any`,
- `std::string_view`,
- `std::filesystem`,
- parallel algorithm support in the standard library,
- `std::byte`,
- `std::from_chars` and `std::to_chars`.

C++20

This revision was one of the largest in language history.
Major additions included:

- concepts,
- ranges,
- coroutines,
- modules,
- three-way comparison,
- designated initializers in C++ form,
- `constexpr` expansion,
- `consteval`,
- `constinit`,
- `std::span`,
- `std::source_location`,
- calendar and time-zone library expansion in `<chrono>`,
- `std::jthread`,
- `std::format`,
- more compile-time and constraints-oriented design power.

C++23

This revision continued the modernization of library design and usability.
Major additions and widely discussed directions included:

- `std::expected`,
- `std::print` and `std::println`,
- `std::mdspan`,
- monadic operations for `std::optional` and `std::expected`,
- ranges refinements and additional views and utilities,
- standard-library growth around usability and expressiveness,
- further improvements to formatting and text-related facilities.

Direction of C++26

Current direction commonly discussed in official status tracking includes ongoing work and implementation interest around topics such as:

- reflection,
- contracts,
- continued library expansion,
- more compile-time capabilities,
- further language cleanup and usability refinements.

Production use should always be tied to verified compiler and standard-library support.

Quick timeline summary

- C++11: the beginning of truly modern everyday C++.
- C++14: refinement and cleanup.
- C++17: major practical usability expansion.
- C++20: very large language and library leap.
- C++23: strong library-centered modernization.
- C++26 direction: more reflection, contracts, and continued evolution.

Professional rule for feature adoption

Do not adopt features by standard name alone. Always check:

- compiler version,
- standard mode,
- active standard library implementation,
- required header or module,
- actual support status on your target environment.

Closing note for the appendices

These appendices are intended as a practical desk reference for repeated use. A professional C++ programmer does not rely only on memory. A professional also keeps close reference material for syntax, common mistakes, command lines, design transitions, and feature history.
Used correctly, these appendices can save time, prevent avoidable mistakes, and strengthen day-to-day confidence while building real Modern C++ software.

References

Purpose of this chapter

This chapter consolidates the most trusted and authoritative sources used throughout this book. The goal is not to provide links, but to clearly identify the **primary sources of truth** for Modern C++:

- the ISO C++ standard,
- official compiler documentation,
- official standard library implementations,
- authoritative reference platforms,
- widely accepted professional guidelines.

A professional C++ developer does not rely on fragmented tutorials alone. Instead, they build the habit of consulting these sources directly.

The ISO C++ Standard

ISO/IEC 14882

The C++ language is formally defined by the ISO/IEC 14882 standard. This document specifies:

- the syntax and grammar of the language,
- object model and memory rules,
- core language semantics,
- standard library requirements,
- compile-time and runtime behavior guarantees,
- definitions of undefined, implementation-defined, and unspecified behavior.

The ISO standard is not designed as a tutorial. It is a formal specification used by compiler implementers and advanced developers.
Professional usage includes:

- verifying exact behavior of edge cases,
- understanding precise rules of templates, lifetimes, and overload resolution,
- confirming guarantees around memory and concurrency,
- resolving ambiguities between different compiler behaviors.

Standard C++ Foundation resources

isocpp.org

This platform provides:

- official news about the C++ standard,
- updates on new revisions such as C++20, C++23, and future directions,
- discussions around language evolution,
- access to community-reviewed resources.

Professional developers use it to stay informed about:

- upcoming features,
- implementation status across compilers,
- standardization progress.

Standard library reference resources

Comprehensive reference platforms

A widely used structured reference for the standard library and language features provides:

- complete coverage of headers and components,
- detailed descriptions of functions, classes, and templates,
- version annotations such as C++11, C++17, C++20, C++23, and C++26,
- examples for most facilities,
- information about complexity guarantees and requirements.

These references include content covering:

- algorithms,
- containers,

- iterators,
- ranges,
- formatting,
- concurrency,
- filesystem,
- numerics,
- memory utilities.

Microsoft C++ documentation (MSVC)

Scope of MSVC documentation

The Microsoft documentation includes:

- language reference aligned with the ISO standard,
- compiler options such as `/std:c++20`, `/permissive-`, and warning levels,
- standard library implementation details,
- runtime libraries,
- platform-specific features,
- intrinsics and architecture-specific extensions.

Professional usage

MSVC documentation is essential when:

- working on Windows-based systems,
- using Visual Studio toolchains,
- understanding compiler-specific diagnostics,
- verifying feature support and extensions,
- working with Windows APIs and integration.

GCC documentation

Key documentation areas

GCC documentation includes:

- compiler command-line options,
- language dialect selection such as `-std=c++20` and `-std=c++23`,
- support status for different standards,
- implementation notes for features,
- GNU extensions,
- standard library reference manuals.

Professional usage

GCC documentation is essential for:

- Linux and cross-platform development,
- understanding compiler flags and optimization behavior,
- verifying feature support in real toolchains,
- working with GNU extensions and dialect modes.

Clang and LLVM documentation

Clang user manual

The Clang user manual describes:

- command-line options,
- language support and extensions,
- diagnostics behavior,
- compilation workflows,
- integration with tooling such as static analysis.

Professional usage

Clang documentation is important when:

- using LLVM-based toolchains,
- working with advanced diagnostics,
- integrating static analysis,
- building tooling based on Clang front-end infrastructure.

Standard library implementations

Major implementations

- Microsoft STL (MSVC),
- libstdc++ (GCC),
- libc++ (Clang/LLVM environments).

Each implementation:

- follows the ISO standard,
- may differ in internal structure,
- may differ in feature maturity,
- may differ in performance characteristics.

Professional implication

Understanding which standard library is in use is critical when:

- using new features such as `std::expected`, `std::print`, or `std::mdspan`,
- debugging platform-specific issues,
- analyzing performance differences,
- ensuring portability.

Compiler manuals and toolchain references

Key topics covered in compiler manuals

- command-line options,
- warning systems,

- optimization levels,
- linking behavior,
- ABI considerations,
- debugging support,
- platform-specific features.

C++ Core Guidelines

Main focus areas

- resource management and RAII,
- ownership models,
- type safety,
- bounds safety,
- concurrency correctness,
- interface design,
- error handling strategies,
- performance-aware design.

Professional usage

The guidelines are used to:

- improve code quality,
- avoid common pitfalls,
- standardize coding practices,
- guide architectural decisions.

Build systems documentation

CMake documentation

CMake is the most widely used cross-platform build system.
Its documentation covers:

- project configuration,

- target-based design,
- dependency management,
- multi-platform builds,
- integration with compilers and IDEs.

Professional usage

Understanding build systems is essential for:

- multi-file and multi-module projects,
- library distribution,
- cross-platform compatibility,
- continuous integration workflows.

Feature tracking and evolution resources

Compiler feature status pages

Compiler projects publish status pages that describe:

- which C++ standard features are implemented,
- which features are experimental,
- which features require special flags,
- which features are incomplete.

Example evolution

- C++20 introduced concepts, ranges, coroutines, and modules,
- C++23 expanded library features such as `std::expected` and printing,
- C++26 is expected to include reflection and contracts,
- compiler support for these features evolves gradually over time.

Using references effectively

A practical workflow

When working on a problem:

1. Identify the feature involved.
2. Check whether it is a language feature or a library feature.
3. Verify the required standard version.
4. Confirm compiler support.
5. Confirm standard library availability.
6. Read the official documentation for exact behavior.

Example workflow

If you want to use `std::expected`:

- verify it is a C++23 feature,
- enable `-std=c++23` or equivalent,
- check whether your standard library supports it,
- compile a minimal test program,
- then integrate it into your project.

Minimal verification example

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

std::expected<int, std::string> parse(int x)
{
    if (x > 0)
        return x;

    return std::unexpected("invalid");
}

int main()
{
    auto r = parse(-1);

    if (r)
```

```
        std::cout << *r << '\n';
    else
        std::cout << r.error() << '\n';
}
```

Compile (Windows examples):

```
cl /std:c++23 /EHsc main.cpp
g++ -std=c++23 main.cpp -o app.exe
clang++ -std=c++23 main.cpp -o app.exe
```

Professional reading strategy

To become professional in C++, reading should follow a structured approach:

- read the standard library reference when using new components,
- read compiler documentation when using flags or diagnosing issues,
- read guidelines when designing systems,
- verify behavior through small experiments,
- maintain personal notes of confirmed behavior.

Final perspective

The references listed in this chapter represent the foundation of professional C++ work.
A strong developer:

- trusts official documentation over assumptions,
- verifies behavior instead of guessing,
- understands the difference between language rules and compiler behavior,
- uses references as daily tools,
- continues learning as the language evolves.

The journey from beginner to professional is not defined by how many features you know, but by how reliably you can use the language with verified knowledge.
This chapter is not the end of the book. It is the beginning of your independent, professional relationship with the C++ ecosystem.