DRAFT

Modern C++ Clean Code

The Definitive Practical Guide
(C++20 & C++23) (First Edition)

Prepared by Ayman Alheraki

Modern C++ Clean Code

The Definitive Practical Guide (C++20 & C++23)

Prepared by Ayman Alheraki

simplifycpp.org

October 2025

Contents

Contents 2

Author's Introduction 12

Introduction 14

The Importance of Clean, Readable, and Maintainable Code 14

Overview of Modern C++ (C++20/23) and Its Main Features 17

ISO Core Guidelines and the Role of Bjarne Stroustrup and Herb Sutter 21

1 Fundamentals of Clean Code 25

1.1 Definition of Clean Code 25

1.1.1 Example: Bad Code vs Clean Code 26

1.2 Golden Rules for Writing Clean Code 29

1.2.1 Example: Bad Code vs Clean Code 30

1.3 Examples of Bad vs. Clean C++ Code 33

1.3.1 Example 1: Looping and Filtering Data 33

1.3.2 Example 2: Memory Management 34

1.3.3 Example 3: Function Design 36

1.3.4 Example 4: Constexpr and Compile-Time Evaluation 37

1.3.5 Key Takeaways 38

1.4 Tools and Practices for Code Quality Verification 40
1.4.1 Static Analysis Tools 40
1.4.2 Unit Testing and Automated Tests 40
1.4.3 Code Review Practices 41
1.4.4 Bad Code vs Clean Code Verification Example 41
1.4.5 Key Takeaways 43

2 Organizing Files and Projects 44
2.1 File and Folder Structure Best Practices 44
2.1.1 Principles of File and Folder Organization 44
2.1.2 Example: Poor vs Proper Structure 46
2.1.3 Best Practices Summary 48
2.2 Proper Use of namespace 49
2.2.1 Principles for Using Namespaces 49
2.2.2 Example: Bad vs Clean Namespace Usage 50
2.2.3 Advanced Example: Nested Namespaces and Aliases 51
2.2.4 Best Practices Summary 52
2.3 Managing #include Directives and Dependencies 54
2.3.1 Principles for Managing Includes 54
2.3.2 Example: Bad vs Clean Include Management 55
2.3.3 Advanced Example: Forward Declaration 56
2.3.4 Best Practices Summary 57
2.4 Modern Project Management with CMake (C++20/23) 59
2.4.1 Principles for Modern CMake Projects 59
2.4.2 Example: Bad vs Clean Project Structure with CMake 60
2.4.3 Key Best Practices for Modern CMake 63

3 Naming Variables and Functions 64
3.1 Modern Naming Conventions for Variables 64
3.1.1 Principles of Modern Variable Naming 64
3.1.2 Example: Bad vs Clean Variable Naming 66
3.1.3 Advanced Example: Loop with Structured Binding 67
3.1.4 Key Takeaways 68
3.2 Expressive Naming for Functions and Classes 69
3.2.1 Principles of Expressive Naming 69
3.2.2 Example: Bad vs Clean Function and Class Naming 70
3.2.3 Advanced Example: Reflecting C++20/23 Features 72
3.2.4 Key Takeaways 73
3.3 CamelCase vs. snake_case 74
3.3.1 Principles for Choosing Naming Conventions 74
3.3.2 Example: Bad vs Clean Usage of Naming Conventions 75
3.3.3 Key Takeaways 77
3.4 Practical Examples from ISO Guidelines 78
3.4.1 ISO Guidelines Principles for Naming 78
3.4.2 Example: Bad vs Clean Code Based on ISO Guidelines 79
3.4.3 Advanced Example: ISO Guidelines with Constants and Scopes . . 81
3.4.4 Key Takeaways from ISO Guidelines 81

4 Program Flow Control 83
4.1 Choosing between if-else, switch, and pattern matching (C++23) 83
4.1.1 Principles for Choosing Flow Control Constructs 83
4.1.2 Example: Bad vs Clean Code 84
4.1.3 Key Takeaways 87
4.2 Exception Handling: try-catch and noexcept 88
4.2.1 Principles of Exception Handling in Modern C++ 88

4.2.2 Example: Bad vs Clean Exception Handling 89
4.2.3 Key Takeaways 91
4.3 Avoiding Complexity and Writing Short, Clear Functions 93
4.3.1 Principles for Short and Clear Functions 93
4.3.2 Example: Bad vs Clean Code 94
4.3.3 Key Takeaways 97

5 Functions and Classes in Modern C++ 99
5.1 Small Functions, Default Parameters, and Use of const and constexpr . . . 99
5.1.1 Principles 99
5.1.2 Example: Bad vs Clean Code 100
5.1.3 Key Takeaways 103
5.2 Designing Clean Classes (Classes/Structs) 104
5.2.1 Principles for Clean Class Design 104
5.2.2 Example: Bad vs Clean Class Design 105
5.2.3 Key Takeaways 109
5.3 Core Principles: RAII, Rule of Five, Rule of Zero 110
5.3.1 RAII (Resource Acquisition Is Initialization) 110
5.3.2 Rule of Five 112
5.3.3 Rule of Zero 115
5.3.4 Key Takeaways 116
5.4 Proper Usage of std::unique_ptr and std::shared_ptr 118
5.4.1 Principles 118
5.4.2 Example: Bad vs Clean Code 119
5.4.3 Advanced Modern C++20/23 Practices 121
5.4.4 Key Takeaways 122

6 Templates and Generic Programming 123
6.1 Writing Safe, Clean Templates . . . 123
6.1.1 Principles of Clean Template Design . . . 123
6.1.2 Example: Bad vs Clean Template . . . 124
6.1.3 Key Takeaways . . . 126
6.2 Using Concepts (C++20) to Improve Clarity of Signatures . . . 128
6.2.1 Principles for Using Concepts . . . 128
6.2.2 Example: Bad vs Clean Template Signatures . . . 129
6.2.3 Key Takeaways . . . 131
6.3 Writing Reusable and Extensible Code . . . 132
6.3.1 Principles for Reusable and Extensible Code . . . 132
6.3.2 Example: Bad vs Clean Code . . . 133
6.3.3 Advanced Example: Extensible Generic Algorithm . . . 134
6.3.4 Key Takeaways . . . 136

7 Modern Programming with STL and Ranges 137
7.1 Effective Use of STL Containers . . . 137
7.1.1 Principles for Effective Use of STL Containers . . . 137
7.1.2 Example: Bad vs Clean Code . . . 138
7.1.3 Key Takeaways . . . 140
7.2 Ranges and Views in C++20 . . . 142
7.2.1 Principles of Using Ranges and Views . . . 142
7.2.2 Example: Bad vs Clean Code . . . 143
7.2.3 Key Takeaways . . . 145
7.3 Writing Clean Loops — Traditional vs Modern Styles . . . 147
7.3.1 Principles for Clean Loops . . . 147
7.3.2 Example: Bad vs Clean Loops . . . 148
7.3.3 Key Takeaways . . . 150

7.4 Examples of Functional-Style Clean Code 152
7.4.1 Principles of Functional-Style Clean Code 152
7.4.2 Example: Bad vs Clean Functional-Style Code 153
7.4.3 Advanced Functional-Style Example 155
7.4.4 Key Takeaways 156

8 Modern Memory Management 157
8.1 Using RAII and Smart Pointers 157
8.1.1 Principles of RAII and Smart Pointers 157
8.1.2 Example: Bad vs Clean Code 158
8.1.3 Key Takeaways 161
8.2 Avoiding Raw Pointers Whenever Possible 163
8.2.1 The Problem with Raw Pointers 163
8.2.2 Modern C++ Philosophy: Ownership Must Be Explicit 163
8.2.3 Example: Bad Code vs Clean Code 164
8.2.4 Best Practices for Clean Memory Management 167
8.2.5 Summary 168
8.3 Resource Management Best Practices 169
8.3.1 The Core Idea: Ownership and Lifetime 169
8.3.2 The Problem: Manual Resource Management 169
8.3.3 Example: Bad Code — Manual Resource Handling 170
8.3.4 Clean Code: RAII and Automatic Scope Control 171
8.3.5 Example: Managing Multiple Resources Safely 172
8.3.6 C++20/23 Best Practices for Resource Management 173
8.3.7 Example: Custom RAII Wrapper (Modern Style) 174
8.3.8 Summary 176

9 Clean Parallelism and Concurrency 177
9.1 Using std::thread and std::async Safely . . . 177
9.1.1 The Problem with Raw Thread Management . . . 177
9.1.2 Clean Code Example – Safe and Structured Thread Usage . . . 178
9.1.3 Leveraging std::async for Simpler Asynchronous Execution . . . 180
9.1.4 Best Practices for Clean Concurrency . . . 181
9.1.5 Summary . . . 182
9.2 Atomic Operations and Mutex – Clean Practices . . . 183
9.2.1 Common Mistakes: Unsafe Shared Data Access . . . 183
9.2.2 Clean Code Example – Using std::atomic for Simple Synchronization 184
9.2.3 Clean Code Example – Using std::mutex and RAII for Shared Data 185
9.2.4 Clean Code Example – std::scoped_lock and Multiple Mutexes . . 188
9.2.5 Best Practices for Clean Synchronization in Modern C++ . . . 189
9.2.6 Summary . . . 189
9.3 Parallel STL Algorithms in C++20/23 . . . 191
9.3.1 The Problem: Manual Thread Management and Complexity . . . 191
9.3.2 Clean Code Example – Using Parallel STL Algorithms . . . 192
9.3.3 Writing Clean Parallel Code Using C++20/23 Ranges . . . 194
9.3.4 Common Mistakes When Using Parallel STL Algorithms . . . 195
9.3.5 Clean Code Example – Exception-Safe Parallel Transformation . . 195
9.3.6 Clean Parallelism Best Practices in Modern C++ . . . 196
9.3.7 Summary . . . 197

10 Modern C++ Features and Best Practices 198
10.1 Concepts, Modules, Coroutines . . . 198
10.1.1 Concepts – Type-Safe and Expressive Templates . . . 198
10.1.2 Modules – Cleaner Project Organization . . . 200
10.1.3 Coroutines – Efficient Asynchronous Programming . . . 202

10.1.4 Best Practices for Clean Modern C++ . . . 203
10.1.5 Summary . . . 204
10.2 std::format, std::span, std::bit_cast . . . 205
10.2.1 std::format – Safe and Readable String Formatting . . . 205
10.2.2 std::span – Safe Views over Arrays and Containers . . . 206
10.2.3 std::bit_cast – Safe Bitwise Type Reinterpretation . . . 208
10.2.4 Best Practices for Clean Modern C++ . . . 209
10.2.5 Summary . . . 209
10.3 Writing Clean, Maintainable Coroutines . . . 210
10.3.1 The Problem: Poorly Structured Coroutines . . . 210
10.3.2 Clean Code Example – Structured, Maintainable Coroutine . . . 211
10.3.3 Best Practices for Clean Coroutines . . . 213
10.3.4 Example: Clean Asynchronous Pipeline with Coroutines . . . 214
10.3.5 Summary . . . 216
10.4 Practical Examples of Modern Features Used Cleanly . . . 217
10.4.1 Combining std::format, std::span, and Ranges . . . 217
10.4.2 Using std::bit_cast for Safe Type Reinterpretation . . . 219
10.4.3 Coroutines for Lazy Generation . . . 219
10.4.4 Combining Concepts, Templates, and Coroutines . . . 222
10.4.5 Key Takeaways . . . 224

11 Testing and Verification 225
11.1 Unit Testing (Catch2, GoogleTest) . . . 225
11.1.1 The Problem: Poorly Written Tests . . . 225
11.1.2 Clean Code Example – Using Catch2 . . . 226
11.1.3 Clean Code Example – Using GoogleTest . . . 228
11.1.4 Best Practices for Clean Unit Tests . . . 229
11.1.5 Example of Clean Test Using Modern C++20/23 . . . 230

11.1.6 Summary 230
11.2 TDD and Writing Testable Code 232
11.2.1 The Problem: Non-Testable Code 232
11.2.2 Clean Code Example – Testable Design 233
11.2.3 Principles for Writing Testable Code 234
11.2.4 Advanced Example: Testable Modern Pipeline 235
11.2.5 Key Takeaways for TDD in Modern C++ 236
11.3 Static Analysis Tools: clang-tidy, cppcheck 236
11.3.1 The Problem: Code Without Static Analysis 236
11.3.2 Clean Code Example – Using clang-tidy 237
11.3.3 Clean Code Example – Using cppcheck 238
11.3.4 Best Practices for Static Analysis 239
11.3.5 Key Takeaways 240
11.4 Static Analysis Tools: clang-tidy, cppcheck 241
11.4.1 The Problem: Code Without Static Analysis 241
11.4.2 Clean Code Example – Using clang-tidy 242
11.4.3 Clean Code Example – Using cppcheck 243
11.4.4 Best Practices for Static Analysis 244
11.4.5 Key Takeaways 245

12 Designing Clean APIs 246
12.1 Guidelines for Designing Clean, Maintainable APIs 246
12.1.1 The Problem: Poorly Designed APIs 246
12.1.2 Clean Code Example – Designing a Safe API 247
12.1.3 Core Guidelines for Clean API Design 249
12.1.4 Advanced Example – Modern, Clean API 250
12.1.5 Key Takeaways 251
12.2 Compatibility and Extensibility 252

12.2.1 The Problem: Rigid, Non-Extensible APIs 252
12.2.2 Clean Code Example – Extensible, Compatible API 253
12.2.3 Guidelines for Compatibility and Extensibility 255
12.2.4 Key Takeaways 256
12.3 Examples from the Standard Library 257
12.3.1 The Problem: Poorly Designed API Usage 257
12.3.2 Clean Code Example – Using Modern Standard Library Features . 258
12.3.3 Lessons from Standard Library API Design 259
12.3.4 Advanced Example – Combining Algorithms and Views 260
12.3.5 Key Takeaways 260

Appendices – Selected ISO Core Guidelines 262
Appendix A: Practical Rules with Examples 262
Appendix B: Interpretation and Practical Application of Each Guideline 269

Conclusion 277
Summary of Best Practices 277
Tips for Developing a Daily Clean Coding Habit 284
Resources and References for Further Learning 289

References 294

Author's Introduction

In today's fast-evolving software industry, the true measure of a skilled developer is no longer the ability to make a program simply work. The real art lies in writing code that is clean, readable, and maintainable — code that not only performs efficiently but also communicates its intent clearly and stands the test of time.

This book, Modern C++ Clean Code: The Definitive Practical Guide (C++20 & C++23), is designed to be your comprehensive and practical roadmap to mastering clean coding principles in the Modern C++ era. It brings together the latest language features introduced in C++20 and C++23, integrating them with the ISO C++ Core Guidelines, a collection of best practices established and refined by the C++ community under the leadership of Bjarne Stroustrup, the creator of the language himself.

Throughout this book, you will find hundreds of detailed, real-world examples illustrating how small design choices can make the difference between fragile, hard-to-maintain code and robust, expressive, and future-proof solutions. Each chapter takes a systematic, professional approach — starting with common bad practices, analyzing their weaknesses, and then refactoring them into clean, modern C++ code.

Beyond syntax and features, this book focuses on clarity, safety, efficiency, and maintainability — the four pillars of professional software craftsmanship. You will learn not only what modern C++ features exist but how and why to use them effectively to design systems that are both elegant and resilient.

Whether you are:

- A professional developer striving to elevate the quality of your codebase,
- A software engineer aiming to align with current industry standards, or
- An educator or student seeking modern, accurate material that reflects the state of C++ today,

this guide will serve as a definitive reference and a daily companion for writing clean, modern, and high-quality C++ code.
Carefully structured and deeply practical, this book represents the culmination of years of experience in Modern C++ design, standardization, and application, bridging the gap between theory and practice. Its purpose is to help you write C++ code that not only works flawlessly but also reads naturally — code that is meant to be understood, not deciphered.
Welcome to Modern C++ — where performance meets clarity, and where clean code becomes the most powerful form of communication between developers.

Stay Connected
For more discussions and valuable content about Modern C++ Clean Code: The Definitive Practical Guide (C++20 & C++23), I invite you to follow me on LinkedIn:
https://linkedin.com/in/aymanalheraki
You can also visit my personal website:
https://simplifycpp.org
Wishing everyone success and prosperity.

Ayman Alheraki

Introduction

The Importance of Clean, Readable, and Maintainable Code

Writing code in C++ is not just about functionality or achieving the desired output. Modern software development demands clean, readable, and maintainable code, especially when working with large-scale systems or collaborative projects. This requirement becomes even more critical with C++20 and C++23, which introduce advanced features such as concepts, ranges, coroutines, modules, and expanded constexpr capabilities. Without a disciplined approach to writing clean code, these features can easily lead to overly complex, unreadable, and error-prone programs. Clean code is characterized by:

1. Readability: Another developer (or even your future self) can understand the code quickly.
2. Maintainability: Changes, bug fixes, or feature additions can be applied with minimal risk of introducing errors.
3. Scalability: Code can evolve as system requirements grow without becoming unmanageable.

Poorly written code may function correctly in the short term but will inevitably accumulate technical debt, making future modifications costly, time-consuming, and

prone to bugs. On the other hand, code that adheres to clean coding principles encourages clarity, consistency, and reliability, allowing teams to fully leverage the expressive power of modern C++.

Example: Bad Code vs Clean Code

Consider a simple scenario of processing a list of integers to compute the sum of all even numbers.
Bad Code (C++20/23 style but unreadable):

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

int main() {
    std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
    int s=0;
    for(auto i:v) if(i%2==0) s+=i;
    std::cout<<s<<"\n";
}
```

Issues in this code:

- The variable names v and s are not descriptive.
- The loop combines iteration and condition in a single line, reducing readability.
- Lacks separation of concerns; computation and output are tightly coupled.

Clean Code (C++20/23 with ranges and clear naming):

```
#include <vector>
#include <ranges>
#include <numeric>
```

```cpp
#include <iostream>

int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5, 6, 7, 8};

    auto even_numbers = numbers | std::ranges::views::filter([](int n) { return n % 2 == 0; });

    int sum_of_evens = std::accumulate(even_numbers.begin(), even_numbers.end(), 0);

    std::cout << "Sum of even numbers: " << sum_of_evens << "\n";
}
```

Improvements in this code:

- Descriptive variable names (numbers, even_numbers, sum_of_evens).
- Uses C++20 ranges for expressive and declarative filtering.
- Computation is clearly separated from output.
- Readable, maintainable, and scalable for future enhancements (e.g., changing the filter or adding more operations).

Key Takeaways:

1. Even small programs benefit from clean naming and structure.
2. Modern C++ features such as ranges, concepts, and coroutines are powerful but can introduce complexity if misused. Clean code principles help harness these features effectively.
3. Writing clean code is an investment that pays off in maintainability, team collaboration, and software longevity.

This section establishes the mindset for the entire booklet: writing C++ code that is not only functional but elegant, readable, and future-proof.

Overview of Modern C++ (C++20/23) and Its Main Features

Modern C++ continues to evolve, offering powerful language features that improve expressiveness, performance, and safety. C++20 and C++23 represent significant milestones, introducing constructs that allow developers to write cleaner, more maintainable, and expressive code. Understanding these features is critical for applying clean code principles effectively.
Key highlights of C++20 include:

- Concepts: Type constraints that make templates easier to understand and safer.
- Ranges and Views: Declarative and composable operations on sequences of data.
- Coroutines: Simplify asynchronous programming and lazy computation.
- Modules: Improve compile times and provide better encapsulation compared to headers.
- constexpr enhancements: Enable more complex computations at compile-time.
- std::format: Type-safe string formatting.
- std::span: Safe, non-owning views of contiguous sequences.

C++23 continues this trend with:

- Expanded standard library algorithms (e.g., std::ranges::zip_view, std::ranges::to).
- std::bit_cast and other utilities for low-level operations safely.
- Improved constexpr support for more complex runtime logic at compile-time.
- Enhanced coroutines and asynchronous programming utilities.

These features make C++ code more declarative, expressive, and self-documenting, provided developers follow clean code practices such as descriptive naming, separation of concerns, and readable flow.

Example: Bad Code vs Clean Code Using Modern C++ Features

Suppose we want to transform a list of integers, filter out odd numbers, square the even numbers, and print the results.
Bad Code (using C++20/23 features poorly):

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

int main() {
    std::vector<int> v{1,2,3,4,5,6};
    for(auto i:v) if(i%2==0) std::cout<<i*i<<' ';
    std::cout<<'\n';
}
```

Issues:

- Variable name v is non-descriptive.
- Logic is compressed into one line, reducing readability.
- Mixing computation and output reduces maintainability.
- Underutilizes the expressive power of ranges.

Clean Code (C++20 with ranges and clear naming):

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

int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5, 6};

    auto even_numbers = numbers
                        | std::ranges::views::filter([](int n) { return n % 2 == 0; })
                        | std::ranges::views::transform([](int n) { return n * n; });

    std::cout << "Squares of even numbers: ";
    for (int square : even_numbers) {
        std::cout << square << ' ';
    }
    std::cout << '\n';
}
```

Improvements:

- Descriptive variable names: numbers, even_numbers.
- Uses C++20 ranges for clear, declarative filtering and transformation.
- Computation is separated from output, improving readability and maintainability.
- Scalable: additional transformations or filters can be added without modifying the loop structure.

Key Takeaways:

1. Modern C++ features are powerful but must be applied thoughtfully to avoid complexity.

2. C++20/23 enables expressive, declarative code, but readability and maintainability are achieved only when clean code principles are applied.

3. Clear separation of computation, transformation, and output is essential for maintainable software.

This section establishes a foundational understanding of what modern C++ offers and demonstrates how clean code practices maximize its potential.

ISO Core Guidelines and the Role of Bjarne Stroustrup and Herb Sutter

The ISO C++ Core Guidelines are a collection of practical rules, best practices, and design principles aimed at writing safe, maintainable, and efficient C++ code. They were primarily designed by Bjarne Stroustrup, the creator of C++, and Herb Sutter, a leading authority on C++ standardization and modern practices. These guidelines provide a foundation for clean code in C++, helping developers leverage advanced features of C++20 and C++23 without introducing undefined behavior, memory safety issues, or unnecessary complexity.
The ISO Core Guidelines emphasize:

1. Resource Safety: Using RAII and smart pointers to manage dynamic memory and resources safely.

2. Expressive Types: Writing code that clearly conveys intent, making use of strong typing and modern constructs.

3. Const-Correctness and Immutability: Ensuring immutability where possible for safer and more predictable code.

4. Simplicity and Readability: Prefer clarity over cleverness; minimize unnecessary complexity.

5. Modern C++ Features: Encourage the use of C++20/23 features, such as ranges, concepts, coroutines, and modules, in a disciplined and maintainable manner.

Stroustrup and Sutter's work guides developers to avoid common pitfalls, such as raw pointer misuse, inconsistent ownership models, and tangled dependencies, all of which can compromise code clarity and maintainability.

Example: Bad Code vs Clean Code Following ISO Core Guidelines

Suppose we want to process a list of dynamically allocated integers, doubling their values and printing the results.
Bad Code (ignores ISO Core Guidelines):

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

int main() {
    std::vector<int*> v;
    for(int i=1;i<=5;++i) v.push_back(new int(i));

    for(int* p : v) std::cout << (*p)*2 << ' ';
    std::cout << '\n';

    // Manual deletion required
    for(int* p : v) delete p;
}
```

Issues:

- Uses raw pointers unnecessarily.
- Manual memory management increases risk of leaks or undefined behavior.
- Variable names (v, p) are non-descriptive.
- Combines computation and output, reducing maintainability.

Clean Code (ISO Core Guidelines, Modern C++20/23):

```
#include <vector>
#include <ranges>
```

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

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

    // Use ranges for transformation and clarity
    auto doubled = numbers
                 | std::ranges::views::transform([](int n) { return n * 2; });

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

Improvements:

- Eliminates raw pointers; uses automatic storage (std::vector<int>).
- Descriptive variable names (numbers, doubled).
- Uses C++20 ranges for clear, declarative transformations.
- No manual resource management; fully adheres to RAII principles.
- Computation is clearly separated from output, improving readability and maintainability.

Key Takeaways:

1. ISO Core Guidelines provide practical rules that align with modern C++20/23 features.

2. Following these guidelines ensures resource safety, maintainability, and clarity, even in complex applications.

3. Stroustrup and Sutter's work serves as a foundation for clean, professional C++ code, enabling teams to write scalable and reliable software without sacrificing expressiveness.

This section introduces the formal principles behind modern C++ clean code and prepares the reader to apply these rules in subsequent chapters, where advanced C++20/23 features are used in professional, maintainable patterns.

Chapter 1

Fundamentals of Clean Code

1.1 Definition of Clean Code

Clean code is code that is clear, readable, maintainable, and expressive, allowing developers to understand and modify it with minimal effort and risk of introducing errors. It is not merely about making code work; it is about writing code that communicates intent explicitly, is structured logically, and is resilient to change.
In the context of C++20 and C++23, clean code emphasizes:

1. Expressiveness: Using language features such as ranges, concepts, coroutines, and constexpr functions to clearly convey purpose.

2. Maintainability: Organizing code so that changes in requirements or logic can be implemented with minimal impact on unrelated parts of the codebase.

3. Safety and Reliability: Leveraging RAII, smart pointers, and type safety to prevent memory leaks, undefined behavior, or concurrency errors.

4. Consistency: Applying naming conventions, code formatting, and design patterns consistently throughout the project.

Writing clean code requires discipline and attention to detail, ensuring that each function, class, and module has a clear responsibility, follows single responsibility principles, and avoids unnecessary complexity.

1.1.1 Example: Bad Code vs Clean Code

Suppose we want to compute the factorial of a number.
Bad Code (C++20/23 but unclear and error-prone):

```
#include <iostream>

int f(int n) {
    if(n<=1) return 1; else return n*f(n-1);
}

int main() {
    int x;
    std::cin >> x;
    std::cout << f(x) << "\n";
}
```

Issues in this code:

- Function name f is non-descriptive.
- Combines conditional logic in one line, reducing readability.
- Input handling is minimal; no explanation of purpose.
- No comments or structure to guide the reader.

Clean Code (C++20/23 with modern practices):

```
#include <iostream>
#include <concepts>

constexpr unsigned long long factorial(unsigned int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}

int main() {
    unsigned int number;
    std::cout << "Enter a non-negative integer: ";
    std::cin >> number;

    auto result = factorial(number);
    std::cout << "Factorial of " << number << " is " << result << "\n";
}
```

Improvements:

- Descriptive function name: factorial.
- Uses constexpr to allow compile-time evaluation when possible.
- Clear input prompt improves readability and user experience.
- Variables have descriptive names (number, result).
- Logic is clean and separated from I/O, improving maintainability.

Key Takeaways:

1. Clean code communicates intent; the reader can understand what the code does without deciphering cryptic names or compressed logic.

2. Modern C++ features (like constexpr, concepts, ranges) enhance readability and expressiveness when applied thoughtfully.

3. Clean code is future-proof: easier to maintain, extend, and integrate into larger systems.

This section establishes the foundational concept of clean code, preparing the reader to explore practical strategies, patterns, and examples in subsequent chapters.

1.2 Golden Rules for Writing Clean Code

Writing clean C++ code requires adherence to a set of fundamental principles that ensure readability, maintainability, and robustness. In the context of C++20 and C++23, these rules help developers harness modern language features without compromising clarity or introducing complexity.
The golden rules include:

1. Descriptive Naming
   Names should communicate purpose clearly. Variables, functions, and classes must reflect their responsibilities. Modern C++ features like concepts and templates make naming even more critical for readability.

2. Single Responsibility Principle (SRP)
   Each function or class should have one clear responsibility. This minimizes side effects and makes testing, debugging, and maintenance easier.

3. Prefer Readability Over Cleverness
   Use advanced features only if they improve clarity. Avoid overly compact or cryptic expressions that obscure intent.

4. Consistent Structure and Formatting
   Maintain a consistent coding style, including indentation, spacing, and brace placement. This improves readability and collaboration in large projects.

5. Use RAII and Smart Pointers
   Always prefer resource-managing constructs (std::unique_ptr, std::shared_ptr) to manage memory safely. Avoid raw pointers unless absolutely necessary.

6. Limit Side Effects

Functions should ideally compute a result without modifying global state. This improves predictability and reduces bugs.

7. Separate Computation from I/O
   Keep algorithmic logic independent of input/output operations to improve testability and maintainability.

8. Leverage Modern Features Safely
   Utilize ranges, coroutines, constexpr, modules, and concepts to write expressive, efficient code, but avoid overcomplicating simple tasks.

1.2.1 Example: Bad Code vs Clean Code

Suppose we want to compute and print all prime numbers up to a given limit.
Bad Code:

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

int main() {
    int n;
    std::cin >> n;
    for(int i=2;i<=n;i++){
        bool p=true;
        for(int j=2;j<i;j++){
            if(i%j==0){p=false;break;}
        }
        if(p) std::cout<<i<<" ";
    }
    std::cout<<"\n";
}
```

Issues:

- Variable names (i, j, p) are unclear.
- Logic is nested and compressed, reducing readability.
- No separation of concerns; computation and output are mixed.
- Not optimized; could leverage modern C++ features.

Clean Code (C++20/23 style):

```
#include <iostream>
#include <vector>
#include <ranges>
#include <cmath>

bool is_prime(unsigned int number) {
    if(number < 2) return false;
    for(unsigned int divisor = 2; divisor <= static_cast<unsigned int>(std::sqrt(number)); ++divisor) {
        if(number % divisor == 0) return false;
    }
    return true;
}

std::vector<unsigned int> generate_primes(unsigned int limit) {
    std::vector<unsigned int> primes(limit);
    std::ranges::iota(primes, 2); // Fill vector with 2..limit
    auto prime_numbers = primes
                         | std::ranges::views::filter(is_prime);
    return std::vector<unsigned int>(prime_numbers.begin(), prime_numbers.end());
}

int main() {
    unsigned int limit;
    std::cout << "Enter the upper limit for primes: ";
```

```
    std::cin >> limit;

    auto primes = generate_primes(limit);

    std::cout << "Prime numbers up to " << limit << ": ";
    for (auto prime : primes) {
        std::cout << prime << ' ';
    }
    std::cout << '\n';
}
```

Improvements:

- Descriptive names: number, divisor, primes, prime_numbers.
- Functions have single responsibilities: is_prime checks primality; generate_primes generates the sequence.
- Uses C++20 ranges and views::filter for clear, declarative filtering.
- Logic is separated from I/O, improving maintainability and testability.
- Computation is optimized by checking divisors only up to $\sqrt{n}$.

Key Takeaways:

1. Descriptive names, separation of concerns, and modern C++ features are central to clean code.
2. Following the golden rules ensures that even complex algorithms remain readable, maintainable, and safe.
3. Clean code is an investment in future-proofing your software, allowing seamless enhancements and easier collaboration.

1.3 Examples of Bad vs. Clean C++ Code

Practical examples are the most effective way to internalize clean code principles. In this section, we demonstrate how minor improvements in naming, structure, and modern C++ features can transform code from difficult-to-maintain to readable, safe, and expressive.

1.3.1 Example 1: Looping and Filtering Data

Bad Code:

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

int main() {
    std::vector<int> v{1,2,3,4,5,6,7,8};
    for(auto i:v)
        if(i%2==0) std::cout<<i*i<<' ';
    std::cout<<'\n';
}
```

Issues:

- Non-descriptive names (v, i).
- Mixing filtering, transformation, and output in one line.
- No separation of concerns.

Clean Code (C++20 Ranges):

```
#include <vector>
#include <ranges>
```

```
#include <iostream>

int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5, 6, 7, 8};

    auto squared_evens = numbers
                        | std::ranges::views::filter([](int n){ return n % 2 == 0; })
                        | std::ranges::views::transform([](int n){ return n * n; });

    std::cout << "Squares of even numbers: ";
    for(int value : squared_evens)
        std::cout << value << ' ';
    std::cout << '\n';
}
```

Improvements:

- Descriptive names: numbers, squared_evens, value.
- Clear separation of filtering, transformation, and output.
- Declarative and expressive use of ranges.

1.3.2 Example 2: Memory Management

Bad Code (using raw pointers):

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

int main() {
    std::vector<int*> data;
    for(int i=1;i<=5;i++) data.push_back(new int(i));
```

```
    for(int* p : data) std::cout << *p << ' ';

    for(int* p : data) delete p; // manual deletion
}
```

Issues:

- Raw pointers introduce risk of leaks.
- Manual memory management is error-prone.
- Poor variable names (data, p).

Clean Code (Modern C++ with RAII):

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

int main() {
    std::vector<std::unique_ptr<int>> data;
    for(int i=1; i<=5; ++i)
        data.push_back(std::make_unique<int>(i));

    std::cout << "Data values: ";
    for(const auto& ptr : data)
        std::cout << *ptr << ' ';
    std::cout << '\n';
}
```

Improvements:

- Automatic memory management via std::unique_ptr.
- Eliminates manual delete calls.
- Descriptive variable names and safe iteration.

1.3.3 Example 3: Function Design

Bad Code (monolithic function):

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

int main() {
    std::vector<int> v{1,2,3,4,5};
    int s=0;
    for(int i : v)
        if(i%2==0) s+=i;
    std::cout<<s<<"\n";
}
```

Issues:

- Functionality is embedded in main.
- Lack of abstraction and separation of responsibility.
- Minimal naming reduces clarity.

Clean Code (Separation and Readability):

```
#include <vector>
#include <ranges>
#include <numeric>
#include <iostream>

int sum_of_even(const std::vector<int>& numbers) {
    auto evens = numbers | std::ranges::views::filter([](int n){ return n % 2 == 0; });
    return std::accumulate(evens.begin(), evens.end(), 0);
}
```

```
int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5};
    int total = sum_of_even(numbers);
    std::cout << "Sum of even numbers: " << total << '\n';
}
```

Improvements:

- Clear function name: sum_of_even.
- Single Responsibility: function computes sum; main handles I/O.
- Uses C++20 ranges for expressive filtering.
- Descriptive variable names improve readability.

1.3.4 Example 4: Constexpr and Compile-Time Evaluation

Bad Code (runtime computation when compile-time possible):

```
#include <iostream>

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

int main() {
    int val = 5;
    std::cout << "Square: " << square(val) << "\n";
}
```

Issues:

- Function could be evaluated at compile-time.

- Missed opportunity to leverage C++20 constexpr for performance and safety.

Clean Code (C++20 constexpr):

```
#include <iostream>

constexpr int square(int n) {
    return n * n;
}

int main() {
    constexpr int val = 5;
    constexpr int result = square(val);
    std::cout << "Square: " << result << "\n";
}
```

Improvements:

- Compile-time evaluation improves performance.
- constexpr communicates intent and improves safety.
- Clear and maintainable naming.

1.3.5 Key Takeaways

1. Modern C++20/23 features such as ranges, smart pointers, and constexpr can significantly improve readability, maintainability, and safety.
2. Clean code emphasizes descriptive naming, separation of concerns, and safe resource management.
3. Small, disciplined improvements transform code from brittle and error-prone to expressive, robust, and future-proof.

This section provides concrete examples of how clean code principles translate directly into practical C++ improvements, setting the foundation for the deeper patterns and techniques explored in the following chapters.

1.4 Tools and Practices for Code Quality Verification

Ensuring code quality is as critical as writing code itself. Modern C++ projects—especially those using C++20 and C++23 features—require tools and practices that verify correctness, maintainability, and adherence to clean coding standards. This includes static analysis, automated testing, and code review practices.

1.4.1 Static Analysis Tools

Static analysis tools inspect code without executing it, detecting potential errors, memory leaks, and violations of coding guidelines. These tools are particularly valuable for modern C++ constructs such as constexpr, coroutines, ranges, and smart pointers.

- clang-tidy: Enforces style rules, detects bugs, and supports modern C++ best practices.
- cppcheck: Focused on error detection, including uninitialized variables and pointer misuse.
- SonarQube: Advanced platform for code quality metrics and analysis.

Best Practice: Integrate static analysis into the build pipeline to ensure continuous verification of code quality.

1.4.2 Unit Testing and Automated Tests

Unit tests validate that individual components behave correctly. Using frameworks such as GoogleTest or Catch2, developers can write tests for functions, classes, and modules, ensuring that C++20/23 features behave as intended.

- Use constexpr functions in tests for compile-time validation.

- Test coroutines and asynchronous code to ensure correct sequencing and results.
- Combine ranges and STL algorithms with unit tests to verify logic correctness.

1.4.3 Code Review Practices

Code reviews enforce consistency, readability, and adherence to clean code principles. Peer reviews are crucial for detecting misuse of modern C++ features, such as unsafe pointer handling, misapplied coroutines, or unnecessary template complexity.
Best Practice:

- Review variable naming, separation of concerns, and function responsibilities.
- Ensure proper use of RAII and smart pointers.
- Validate that modern C++ features improve clarity rather than obscure logic.

1.4.4 Bad Code vs Clean Code Verification Example

Suppose we want to create a simple function that calculates the sum of squares of even numbers.
Bad Code:

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

int f(std::vector<int> v) {
    int s=0;
    for(auto i:v) if(i%2==0) s+=i*i;
    return s;
}
```

```
int main() {
    std::vector<int> nums{1,2,3,4,5};
    std::cout << f(nums) << "\n";
}
```

Issues:

- Function name f is unclear.
- Single-letter variable names reduce readability.
- Logic compressed in one line makes static analysis and code review harder.
- Lacks proper unit testing to verify correctness.

Clean Code (C++20/23, with testable, verifiable design):

```
#include <vector>
#include <ranges>
#include <numeric>
#include <iostream>

constexpr int sum_of_squares_of_even(const std::vector<int>& numbers) {
    auto evens = numbers | std::ranges::views::filter([](int n){ return n % 2 == 0; })
                         | std::ranges::views::transform([](int n){ return n * n; });
    return std::accumulate(evens.begin(), evens.end(), 0);
}

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

    int result = sum_of_squares_of_even(numbers);
    std::cout << "Sum of squares of even numbers: " << result << "\n";
}
```

Verification Advantages:

- Descriptive function and variable names improve static analysis results.
- Clear separation of computation and I/O facilitates unit testing.
- Using ranges and transform views makes logic explicit, reducing errors.
- constexpr allows compile-time evaluation, providing early detection of logical errors.

1.4.5 Key Takeaways

1. Modern C++20/23 features require careful verification to ensure correct, maintainable, and safe code.
2. Static analysis, automated testing, and peer reviews are essential for upholding clean code principles.
3. Structuring code for clarity, testability, and analyzability improves long-term maintainability and reliability.

This section establishes a practical approach to verifying the quality of modern C++ code, emphasizing that clean code is not just stylistic, but verifiably correct and robust.

Chapter 2

Organizing Files and Projects

2.1 File and Folder Structure Best Practices

A well-organized file and folder structure is fundamental to maintaining clean, scalable, and maintainable C++ projects. Proper organization improves readability, collaboration, and build efficiency, and is essential when working with modern C++20/23 features like modules, concepts, and coroutines.

2.1.1 Principles of File and Folder Organization

1. Separate Interface from Implementation
    - Header files (.h or .hpp) should declare interfaces.
    - Source files (.cpp) should define implementation details.
    - With C++20 modules, consider .ixx for module interfaces and .cppm for implementation units.
2. Logical Grouping of Functionality

- Group related classes, functions, and modules in dedicated folders.
- Examples:
  - core/ for central algorithms or engine code.
  - utils/ for helper functions and utilities.
  - io/ for input/output handling.

3. Consistent Naming Conventions

- Use camelCase or snake_case consistently.
- File names should reflect their contents: e.g., file_manager.hpp, vector_utils.cpp.

4. Minimal Coupling Between Modules

- Avoid deeply nested dependencies between folders.
- Use forward declarations and interfaces to reduce coupling.

5. Include Guards or #pragma once

- Always protect headers against multiple inclusion.
- C++20 modules reduce the need for traditional include guards.

6. Test and Examples Separation

- Maintain separate folders for tests (tests/) and sample programs (examples/).
- Keep production code independent from testing infrastructure.

2.1.2 Example: Poor vs Proper Structure

Bad Structure: Single folder, everything in main.cpp

```
project/

   main.cpp
```

main.cpp:

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

int main() {
    std::vector<int> nums{1,2,3,4,5};
    int sum=0;
    for(auto n: nums) sum+=n;
    std::cout<<"Sum: "<<sum<<"\n";
}
```

Issues:

- No separation of responsibilities.
- Core logic, utilities, and main program are all in one file.
- Difficult to maintain as project grows or when adding tests.

Clean Structure: Modular and Organized

```
project/
   src/
      main.cpp
```

```
    utils/
        math_utils.hpp
        math_utils.cpp
    core/
        calculator.hpp
        calculator.cpp
tests/
    test_calculator.cpp
CMakeLists.txt
```

src/utils/math_utils.hpp

```cpp
#pragma once
#include <vector>

int sum(const std::vector<int>& numbers);
```

src/utils/math_utils.cpp

```cpp
#include "math_utils.hpp"
#include <numeric>

int sum(const std::vector<int>& numbers) {
    return std::accumulate(numbers.begin(), numbers.end(), 0);
}
```

src/main.cpp

```cpp
#include <iostream>
#include "utils/math_utils.hpp"

int main() {
    std::vector<int> numbers{1, 2, 3, 4, 5};
    std::cout << "Sum: " << sum(numbers) << "\n";
}
```

Improvements:

- Separation of concerns: sum utility is independent of main.
- Files are grouped logically (utils/, core/).
- Future additions (like more math functions or tests) are easy to integrate.
- Ready for C++20 modules if needed: math_utils could become a module interface.

2.1.3 Best Practices Summary

1. Group related code in folders to improve modularity and readability.
2. Separate headers and implementation; use modules for modern C++20/23 projects.
3. Adopt clear and consistent naming conventions.
4. Isolate tests and examples from production code.
5. Minimize coupling and prefer interfaces for flexibility and maintainability.

Organizing files and folders effectively is a foundational step toward clean, maintainable C++ projects, enabling teams to scale codebases safely and efficiently while fully leveraging modern C++20 and C++23 features.

2.2 Proper Use of namespace

Namespaces in C++ are essential tools for organizing code, avoiding name collisions, and clarifying intent. Modern C++20/23 code, which often leverages modules, templates, and third-party libraries, relies on well-structured namespaces to maintain clarity and maintainability.

2.2.1 Principles for Using Namespaces

1. Avoid Global Scope Pollution
    - Do not place variables, functions, or classes in the global namespace unless absolutely necessary.
    - Global scope increases the risk of name collisions, especially when using third-party libraries.
2. Logical Grouping of Related Code
    - Use namespaces to group classes, functions, and constants that logically belong together.
    - Example: math, io, core, utils.
3. Nested Namespaces for Hierarchical Structure
    - C++17 and later support nested namespace syntax:

      ```
      namespace project::math { /* ... */ }
      ```

    - This provides clarity while reducing verbose indentation.
4. Avoid using namespace in Headers

- Never place using namespace in header files; it forces all including files into that namespace.
- Prefer qualified names or using directives in local scopes (e.g., inside functions).

5. Short Aliases for Long Namespaces

- For deeply nested or lengthy namespaces, define an alias inside implementation files:

```
namespace pm = project::math;
```

2.2.2 Example: Bad vs Clean Namespace Usage

Bad Code (global pollution and unclear organization):

```
#include <iostream>

int value = 42;

void print_value() {
    std::cout << value << "\n";
}

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

Issues:

- value and print_value exist in the global namespace.
- Risk of collision if another library or module defines the same names.

- No logical grouping, making code harder to maintain in large projects.

Clean Code (Proper Namespaces, C++20/23 style):

```
#include <iostream>

namespace project::core {

    int value = 42;

    void print_value() {
        std::cout << value << "\n";
    }

} // namespace project::core

int main() {
    project::core::print_value(); // clearly shows the origin of the function
}
```

Improvements:

- Variables and functions are contained in project::core.
- Clear separation avoids collisions.
- Code communicates logical ownership and is easier to extend.

2.2.3 Advanced Example: Nested Namespaces and Aliases

```
#include <iostream>

namespace project::math::geometry {
```

```
    constexpr double pi = 3.141592653589793;

    double circle_area(double radius) {
        return pi * radius * radius;
    }

} // namespace project::math::geometry

int main() {
    namespace geom = project::math::geometry;
    double r = 5.0;

    std::cout << "Area of circle: " << geom::circle_area(r) << "\n";
}
```

Improvements:

- Nested namespace clearly communicates hierarchical organization.
- Alias geom reduces verbosity while keeping clarity.
- Constants (pi) and functions (circle_area) are logically grouped.

2.2.4 Best Practices Summary

1. Use namespaces to group logically related code, minimizing global scope pollution.
2. Avoid using namespace in headers; use it locally in implementation files if needed.
3. Leverage nested namespaces to reflect hierarchical design.
4. Create aliases for long namespaces to improve readability.
5. Align namespace organization with folder and module structure for maximum clarity.

Proper namespace usage is a cornerstone of clean, maintainable C++ code, ensuring that even large projects with multiple modules, libraries, and contributors remain organized, readable, and collision-free in modern C++20/23 development.

2.3 Managing #include Directives and Dependencies

Efficient management of #include directives and dependencies is critical for clean, maintainable, and fast-compiling C++ projects, especially in modern C++20 and C++23 development. Poor management can lead to slow builds, circular dependencies, and hard-to-debug errors.

2.3.1 Principles for Managing Includes

1. Include What You Use (IWYU)
    - Only include headers that are necessary for the current file.
    - Avoid indirect inclusions that rely on headers included elsewhere.
2. Use Forward Declarations When Possible
    - Forward declarations reduce compilation time and minimize dependencies.
    - Example: class MyClass; instead of #include ”MyClass.hpp” in headers when only pointers or references are needed.
3. Prefer <...> for Standard Library and ”” for Project Headers
    - Use angle brackets for system or standard library headers.
    - Use quotes for your project headers.
4. Minimize Header Interdependencies
    - Avoid including headers unnecessarily inside other headers.
    - Favor implementation files (.cpp) for including full definitions.

5. Leverage Modules (C++20/23)

    - Modern C++ modules reduce unnecessary text inclusion and speed up compilation.
    - Modules provide explicit interfaces, replacing many traditional headers.

6. Include Guards or #pragma once

    - Always protect headers from multiple inclusion to prevent redefinition errors.
    - Modules eliminate this need, but legacy headers still require guards.

2.3.2 Example: Bad vs Clean Include Management

Bad Code: Excessive and unnecessary includes
math_utils.hpp:

```
#pragma once
#include <vector>
#include <iostream>
#include <algorithm>
#include <numeric>
#include <string>
#include <map> // unused

int sum(const std::vector<int>& numbers) {
    return std::accumulate(numbers.begin(), numbers.end(), 0);
}
```

Issues:

- Includes headers that are not needed (<string>, <map>).

- Mixing implementation in header (sum defined in header) increases coupling.
- Compilation time increases for larger projects.

Clean Code (Minimal includes and proper separation)
math_utils.hpp:

```
#pragma once
#include <vector> // only needed header

int sum(const std::vector<int>& numbers); // function declaration only
```

math_utils.cpp:

```
#include "math_utils.hpp"
#include <numeric> // only needed in implementation

int sum(const std::vector<int>& numbers) {
    return std::accumulate(numbers.begin(), numbers.end(), 0);
}
```

Improvements:

- Header contains only declarations and minimal includes.
- Implementation file includes <numeric> as required.
- Reduces compilation dependencies and improves maintainability.

2.3.3 Advanced Example: Forward Declaration

Bad Code: Including unnecessary header

```
#include ”calculator.hpp” // full include
#include <iostream>

void print_sum(const Calculator& calc) {
    std::cout << calc.total() << ”\n”;
}
```

Clean Code: Using forward declaration

```
#include <iostream>

class Calculator; // forward declaration

void print_sum(const Calculator& calc);
```

Advantages:

- Reduces coupling and compilation time.
- Avoids unnecessary includes in headers.
- Clean separation of interface and implementation.

2.3.4 Best Practices Summary

1. Include only what you use; avoid indirect and redundant headers.
2. Use forward declarations in headers when possible.
3. Keep implementation files responsible for full includes.
4. Protect headers with include guards or #pragma once.
5. Consider C++20/23 modules for large projects to reduce dependency complexity.

6. Maintain consistent order: standard library, third-party, then project headers.

Proper management of #include directives ensures faster builds, fewer compilation errors, and clean modular design. When combined with namespaces and folder structure, it significantly enhances the maintainability and scalability of modern C++20/23 projects.

2.4 Modern Project Management with CMake (C++20/23)

Modern C++ projects benefit significantly from structured build systems, and CMake is the de facto standard for managing C++20 and C++23 codebases. Proper CMake configuration ensures modular builds, easy integration of libraries, reproducibility, and clean dependency management.

2.4.1 Principles for Modern CMake Projects

1. Use Target-Based Commands
    - Prefer add_library and add_executable combined with target_include_directories, target_compile_features, and target_link_libraries.
    - Avoid global include directories or compile flags.
2. Specify C++ Standard Explicitly

```
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

3. Organize Subdirectories Logically
    - Split project into src/, include/, tests/, and optional examples/.
    - Each subdirectory can have its own CMakeLists.txt for modular builds.
4. Encapsulate Dependencies
    - Use modern CMake targets instead of global variables.

- Interface libraries (INTERFACE) allow propagation of include paths and compile options safely.

5. Support Modern C++ Features

   - Enable modules and proper compile flags for coroutines, concepts, ranges, and constexpr functions.
   - Use CMake features to enforce warnings as errors and maintain code quality.

6. Separation of Third-Party Libraries

   - Use FetchContent or find_package to integrate external libraries cleanly without polluting global scope.

2.4.2 Example: Bad vs Clean Project Structure with CMake

Bad Code: Monolithic CMakeLists.txt

```
project/

  main.cpp
  math_utils.hpp
  math_utils.cpp
  CMakeLists.txt
```

CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

add_executable(MyProject main.cpp math_utils.cpp)
include_directories(.)
set(CMAKE_CXX_STANDARD 20)
```

Issues:

- All files compiled in one target without modularity.
- include_directories(.) pollutes global scope.
- Hard to scale for larger projects, tests, or submodules.

Clean Code: Modular Project Structure

```
project/
  src/
    CMakeLists.txt
    main.cpp
    utils/
      CMakeLists.txt
      math_utils.hpp
      math_utils.cpp
  tests/
    CMakeLists.txt
  include/
    utils/
      math_utils.hpp
  CMakeLists.txt
```

Top-Level CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.22)
project(MyProject LANGUAGES CXX)

# Require modern C++
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

```
# Add subdirectories
add_subdirectory(src)
add_subdirectory(tests)
```

src/CMakeLists.txt:

```
# Create library for utils
add_library(utils
    utils/math_utils.cpp
)

target_include_directories(utils
    PUBLIC ${CMAKE_SOURCE_DIR}/include
)

# Add executable and link utils
add_executable(MyProject main.cpp)
target_link_libraries(MyProject PRIVATE utils)
```

Improvements:

- Modular build: libraries and executables separated.
- Clear interface include paths for maintainability.
- C++23 standard explicitly enforced.
- Scales easily for additional modules, tests, and examples.
- Enables modern C++ features and compiler options per target.

2.4.3 Key Best Practices for Modern CMake

1. Organize code into libraries and executables to reduce coupling.
2. Use target_* commands instead of global variables.
3. Enforce C++20/23 standards and compiler warnings for clean, modern builds.
4. Keep include directories minimal and target-specific.
5. Separate tests, examples, and third-party libraries.
6. Plan scalable folder structure to accommodate project growth.

Modern CMake ensures clean, maintainable, and scalable project management, allowing developers to leverage C++20/23 features efficiently, reduce compilation overhead, and maintain a robust development workflow.

Chapter 3

Naming Variables and Functions

3.1 Modern Naming Conventions for Variables

Variable naming is a fundamental aspect of clean code, directly affecting readability, maintainability, and clarity of intent. In modern C++20/23, which heavily relies on templates, ranges, and structured bindings, clear and consistent naming is critical for both human understanding and safe collaboration in large projects.

3.1.1 Principles of Modern Variable Naming

1. Descriptive and Unambiguous

    - Variable names should clearly indicate purpose and type.
    - Avoid generic names like tmp, data, or val unless their scope is extremely limited.

2. Consistency Across the Codebase

- Choose a convention (camelCase, snake_case) and maintain it consistently.
- Example: totalCount (camelCase) vs total_count (snake_case).

3. Indicate Variable Scope or Lifetime When Relevant

- Use prefixes or suffixes to clarify pointer ownership, references, constants, or member variables.
- Common conventions:
  - m_ for member variables: m_totalCount
  - s_ for static variables: s_instanceCount
  - g_ for global variables (use sparingly)

4. Use Plural for Collections

- Collections or arrays should be plural to indicate multiple elements:

```
std::vector<int> numbers;
```

5. Avoid Hungarian Notation

- Modern C++ emphasizes strong typing, so encoding type in the variable name is redundant.
- Rely on descriptive names and type hints from IDEs or compiler diagnostics.

6. Prefer Readable Short Names in Local Scope

- For small, localized scopes (loops, lambdas), short names are acceptable: i, j, n.

3.1.2 Example: Bad vs Clean Variable Naming

Bad Code:

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

int main() {
    std::vector<int> v{1,2,3,4,5};
    int s = 0;
    for(auto i : v) s+=i;
    int t = s * 2;
    std::cout << "Result: " << t << "\n";
}
```

Issues:

- Variable names v, s, i, t are ambiguous.
- Purpose of each variable is unclear, reducing maintainability.
- Hard to extend code or integrate with other modules.

Clean Code (Descriptive and Modern C++20 style):

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

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

    // Sum of all numbers
```

```
    int totalSum = std::accumulate(numbers.begin(), numbers.end(), 0);

    // Double the total sum
    int doubledSum = totalSum * 2;

    std::cout << "Doubled sum: " << doubledSum << "\n";
}
```

Improvements:

- Descriptive names: numbers, totalSum, doubledSum.
- Clear separation of logic and intent.
- Easier to debug, extend, and maintain.

3.1.3 Advanced Example: Loop with Structured Binding

Modern C++20 supports structured bindings and ranges, which benefit from clear variable naming:

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

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

    for (int number : numbers | std::ranges::views::filter([](int n){ return n % 2 == 0; })) {
        int square = number * number;
        std::cout << "Square of " << number << ": " << square << "\n";
    }
}
```

Advantages:

- number clearly represents the element of iteration.
- square describes the result of computation.
- Readable and self-documenting, especially in modern pipelines using ranges and lambdas.

3.1.4 Key Takeaways

1. Use descriptive, unambiguous names to express intent.
2. Maintain consistent naming conventions across the project.
3. Reflect scope and lifetime in variable names when necessary.
4. Use plural for collections and readable short names for localized loops.
5. Modern C++20/23 features like ranges, structured bindings, and lambdas benefit significantly from clear variable naming.

Consistent and meaningful variable naming is a cornerstone of clean, maintainable modern C++ code, making code easier to read, understand, and extend while leveraging the full power of C++20/23 features.

3.2 Expressive Naming for Functions and Classes

In modern C++20/23, functions and classes are the primary building blocks of abstraction. Choosing expressive, meaningful names for them is critical for readability, maintainability, and safe usage. Proper naming communicates intent without requiring excessive comments and reduces misunderstandings in large codebases.

3.2.1 Principles of Expressive Naming

1. Describe What, Not How
    - Function names should reflect what the function does, not how it performs the task.
    - Example: calculateTotal() instead of loopAndSum().
2. Use Verb-Noun Pattern for Functions
    - Functions perform actions; include a verb to indicate behavior.
    - Example: loadConfiguration(), sendMessage(), sortNumbers().
3. Use Nouns for Classes
    - Class names should represent concepts or entities.
    - Example: VectorCalculator, FileManager, HttpRequest.
4. Avoid Ambiguous Abbreviations
    - Abbreviations reduce readability and clarity.
    - Prefer configurationManager instead of cfgMgr.

5. Maintain Consistency Across the Codebase

   - Follow the same style for verbs, nouns, and capitalization (camelCase or PascalCase).
   - Use PascalCase for classes: MyClass, MathUtils.
   - Use camelCase for functions: computeAverage(), printResults().

6. Reflect Modern C++20/23 Concepts

   - Leverage concepts, coroutines, and ranges in function and class naming to reflect behavior.
   - Example: filterEvenNumbers() clearly conveys the use of a filtering operation.

3.2.2 Example: Bad vs Clean Function and Class Naming

Bad Code:

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

class Calc {
public:
    int f(std::vector<int> v) {
        int s = 0;
        for(auto i : v) s += i;
        return s;
    }
};

int main() {
    Calc c;
```

```
    std::vector<int> nums{1,2,3,4,5};
    std::cout << c.f(nums) << "\n";
}
```

Issues:

- Class name Calc is too generic.
- Function name f is meaningless.
- Variable names v and s reduce readability.
- Hard to maintain or understand for new developers.

Clean Code (Descriptive and Modern C++20 style):

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

class VectorCalculator {
public:
    int sumElements(const std::vector<int>& numbers) const {
        int total = 0;
        for(int number : numbers) total += number;
        return total;
    }
};

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

    int totalSum = calculator.sumElements(numbers);
    std::cout << "Total sum: " << totalSum << "\n";
}
```

Improvements:

- Class name VectorCalculator clearly indicates purpose.
- Function name sumElements describes the exact action.
- Parameter numbers and local variable total improve readability.
- Code is self-documenting, reducing the need for additional comments.

3.2.3 Advanced Example: Reflecting C++20/23 Features

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

class NumberFilter {
public:
    std::vector<int> filterEvenNumbers(const std::vector<int>& numbers) const {
        std::vector<int> evens;
        for(int number : numbers | std::ranges::views::filter([](int n){ return n % 2 == 0; })) {
            evens.push_back(number);
        }
        return evens;
    }
};

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

    auto evenNumbers = filter.filterEvenNumbers(numbers);
    for(int n : evenNumbers) std::cout << n << " ";
}
```

Advantages:

- NumberFilter class and filterEvenNumbers() function are intuitive and descriptive.
- Code communicates behavior aligned with modern C++20 ranges and functional patterns.
- Improves maintainability, testability, and clarity.

3.2.4 Key Takeaways

1. Use descriptive, unambiguous names for classes and functions.
2. Follow verb-noun patterns for functions and nouns for classes.
3. Avoid vague abbreviations; prefer clarity over brevity.
4. Align naming conventions with C++20/23 features such as ranges, coroutines, and concepts.
5. Consistency across the project improves readability, collaboration, and long-term maintainability.

Expressive naming in modern C++ is a cornerstone of clean code, ensuring that developers can understand, extend, and maintain code effortlessly, even in complex systems leveraging C++20/23 advanced features.

3.3 CamelCase vs. snake_case

Choosing a consistent naming convention is essential for readability, maintainability, and collaboration in modern C++20/23 projects. Two of the most widely used conventions are camelCase and snake_case, each with advantages depending on context and project style guidelines.

3.3.1 Principles for Choosing Naming Conventions

1. Consistency is Key
    - Select one convention per project or per type (variables, functions, classes) and use it consistently.
    - Inconsistent naming increases cognitive load and reduces readability.
2. CamelCase
    - Starts with a lowercase letter for variables/functions (myVariable, calculateSum).
    - UpperCamelCase (PascalCase) for class names (VectorCalculator, NumberFilter).
    - Common in modern C++ codebases, libraries, and APIs.
3. snake_case
    - Lowercase words separated by underscores (my_variable, calculate_sum).
    - Often used in cross-platform projects, C APIs, or embedded systems.
    - Favored for constants in some style guides: constexpr double pi_value = 3.1415;.

4. Scope-Based Differentiation

    - Some projects use camelCase for local variables and functions, snake_case for global variables, macros, or constants.
    - Example: MAX_BUFFER_SIZE for compile-time constants.

5. Align With Standard Libraries

    - Standard library (STL) mostly uses snake_case in function names and types like begin(), end().
    - Mixing conventions may confuse readers.

3.3.2 Example: Bad vs Clean Usage of Naming Conventions

Bad Code: Mixed and inconsistent naming

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

int TotalSum = 0;
std::vector<int> numbersVector{1,2,3,4,5};

for(auto num : numbersVector) TotalSum += num;

std::cout << "Total Sum: " << TotalSum << "\n";
```

Issues:

- TotalSum uses PascalCase for a variable.
- numbersVector combines camelCase and descriptive type name, creating redundancy.

- Inconsistent style reduces readability and increases cognitive overhead.

Clean Code: Consistent camelCase convention

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

int totalSum = 0;
std::vector<int> numbers{1, 2, 3, 4, 5};

for (int number : numbers) totalSum += number;

std::cout << "Total sum: " << totalSum << "\n";
```

Improvements:

- Variables use consistent camelCase.
- Clear, concise names (numbers, totalSum, number) reflect purpose without redundancy.
- Readable, maintainable, and aligned with modern C++ conventions.

Alternative Clean Code: snake_case for constants and global variables

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

constexpr int max_buffer_size = 1024;
std::vector<int> numbers{1, 2, 3, 4, 5};

int total_sum = 0;
for (int number : numbers) total_sum += number;

std::cout << "Total sum: " << total_sum << "\n";
```

Advantages:

- snake_case emphasizes constants and global variables.
- Separates global scope elements from local camelCase variables.
- Maintains readability and reduces confusion in large projects.

3.3.3 Key Takeaways

1. Choose one convention and maintain it consistently across variables, functions, and classes.
2. CamelCase is often preferred for local variables, functions, and classes.
3. snake_case is suitable for constants, macros, or cross-platform code.
4. Avoid mixing styles in the same scope or file to reduce confusion.
5. Align naming with modern C++20/23 idioms and standard libraries to improve readability and maintainability.

Consistent use of camelCase or snake_case improves clarity, team collaboration, and maintainability, especially in modern C++20/23 projects with advanced features such as ranges, coroutines, concepts, and modules.

3.4 Practical Examples from ISO Guidelines

The ISO C++ Core Guidelines, authored by Bjarne Stroustrup and Herb Sutter, provide a rigorous foundation for writing clear, safe, and maintainable C++ code. One critical area they emphasize is naming conventions, which improve readability, reduce errors, and facilitate long-term maintenance in modern C++20/23 projects.

3.4.1 ISO Guidelines Principles for Naming

1. Names Should Convey Meaning
    - Choose names that clearly describe the purpose of variables, functions, and classes.
    - ISO Guideline: "N.1: Names should reveal intent."
2. Avoid Cryptic Abbreviations
    - Use descriptive names even if longer.
    - Example: prefer currentIndex over ci.
3. Use Consistent Style
    - Use camelCase for variables and functions and PascalCase for classes, or follow a team-wide convention.
4. Differentiate Between Types and Values
    - Class names should be nouns; function names should be verbs.
    - Constants should indicate immutability using ALL_CAPS or constexpr with descriptive names.

5. Scope Awareness

    - Use naming to clarify ownership or lifetime, e.g., m_ for members, s_ for static members.

3.4.2 Example: Bad vs Clean Code Based on ISO Guidelines

Bad Code:

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

class C {
public:
    int f(std::vector<int> v) {
        int s = 0;
        for(auto i : v) s += i;
        return s;
    }
};

int main() {
    C c;
    std::vector<int> v{1,2,3,4,5};
    std::cout << c.f(v) << "\n";
}
```

Issues:

- Class name C is meaningless.
- Function f does not reveal intent.
- Variable names v, s, and i are cryptic.

- Hard to maintain, extend, or understand.

Clean Code (Following ISO Guidelines):

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

class VectorCalculator {
public:
    // Sum all elements in the vector
    int sumElements(const std::vector<int>& numbers) const {
        int totalSum = 0;
        for(int number : numbers) totalSum += number;
        return totalSum;
    }
};

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

    int totalSum = calculator.sumElements(numbers);
    std::cout << "Total sum: " << totalSum << "\n";
}
```

Improvements:

- VectorCalculator clearly represents the concept and purpose.
- sumElements describes what the function does.
- numbers and totalSum are descriptive and readable.
- Conforms to ISO guideline: names reveal intent and maintain consistency.

3.4.3 Advanced Example: ISO Guidelines with Constants and Scopes

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

class CircleCalculator {
public:
    static constexpr double PI = 3.141592653589793;

    double calculateArea(double radius) const {
        return PI * radius * radius;
    }
};

int main() {
    CircleCalculator calculator;
    double radius = 5.0;

    double area = calculator.calculateArea(radius);
    std::cout << "Circle area: " << area << "\n";
}
```

ISO Guideline Advantages:

- CircleCalculator is a descriptive class name.
- calculateArea is an expressive verb-noun function name.
- Constant PI is constexpr and uppercase, reflecting immutable values.
- Clear separation of concepts, actions, and constants, improving maintainability.

3.4.4 Key Takeaways from ISO Guidelines

1. Names should reveal intent and avoid ambiguity.

2. Use consistent conventions for variables, functions, classes, and constants.

3. Functions should be verb-oriented, classes noun-oriented, constants uppercase or constexpr descriptive names.

4. Scope and lifetime can be reflected through prefixes like m_ or s_, when appropriate.

5. Applying ISO C++ Guidelines ensures code is readable, maintainable, and aligned with modern C++20/23 standards.

Following ISO Core Guidelines for naming enables self-documenting, safe, and maintainable C++ code, making projects more robust, readable, and scalable in the era of C++20/23 with ranges, coroutines, concepts, and modules.

Chapter 4

Program Flow Control

4.1 Choosing between if-else, switch, and pattern matching (C++23)

In modern C++20/23, program flow control is more expressive and powerful than ever. Choosing the appropriate construct—if-else, switch, or C++23 pattern matching—is essential for readable, maintainable, and clean code.

4.1.1 Principles for Choosing Flow Control Constructs

1. Use if-else for Conditions Involving Ranges or Complex Logic

    - Suitable for non-discrete values, comparisons, or compound logical expressions.
    - Example: checking ranges, floating-point comparisons, or multiple conditions.

2. Use switch for Discrete Integral Values

- Optimized for enum types or integers.
- Enables compiler optimization and clear branching for discrete cases.
- Avoid large switch statements with duplicated code; refactor if necessary.

3. Use Pattern Matching (C++23) for Type-Safe and Complex Cases

- Supports structured bindings, type patterns, and value matching.
- Provides safer and more expressive alternatives to chained if-else.
- Reduces boilerplate code when handling variant types, structs, or enums.

4. Readability and Maintainability First

- Avoid deeply nested if-else chains.
- Prefer single-responsibility branching for clarity.

4.1.2 Example: Bad vs Clean Code

Bad Code: Chained if-else for enum values

```
#include <iostream>

enum class Color { Red, Green, Blue };

void printColor(Color c) {
    if (c == Color::Red) std::cout << "Red\n";
    else if (c == Color::Green) std::cout << "Green\n";
    else if (c == Color::Blue) std::cout << "Blue\n";
    else std::cout << "Unknown\n";
}

int main() {
```

```
    printColor(Color::Green);
}
```

Issues:

- Multiple if-else branches reduce readability.
- Hard to extend for additional enum values.
- Verbose compared to a switch or pattern matching approach.

Clean Code: Using switch (C++20/23)

```
#include <iostream>

enum class Color { Red, Green, Blue };

void printColor(Color c) {
    switch(c) {
        case Color::Red:   std::cout << "Red\n"; break;
        case Color::Green: std::cout << "Green\n"; break;
        case Color::Blue:  std::cout << "Blue\n"; break;
    }
}

int main() {
    printColor(Color::Green);
}
```

Improvements:

- Clear, concise, and optimized for discrete enum values.
- Easier to maintain and extend.

- Reduces cognitive load and eliminates unnecessary comparisons.

Advanced Clean Code: Using Pattern Matching (C++23)

```
#include <variant>
#include <iostream>

using Color = std::variant<int, std::string>; // Example variant type

void printColor(const Color& c) {
    if (auto colorValue = std::get_if<int>(&c)) {
        std::cout << "Integer color code: " << *colorValue << "\n";
    } else if (auto colorName = std::get_if<std::string>(&c)) {
        std::cout << "Color name: " << *colorName << "\n";
    }
}

int main() {
    Color color1 = 1;
    Color color2 = std::string("Blue");

    printColor(color1);
    printColor(color2);
}
```

Advantages of Pattern Matching:

- Type-safe handling of variant types.
- Reduces the risk of invalid assumptions or runtime errors.
- Clear separation of different types and structures.
- Aligns with modern C++23 expressive control flow best practices.

4.1.3 Key Takeaways

1. if-else: Best for complex conditions or ranges.
2. switch: Best for discrete integral or enum values.
3. Pattern matching (C++23): Best for variant types, structured data, and type-safe branching.
4. Always prioritize readability, maintainability, and safety.
5. Refactor nested or repetitive flow into smaller functions or pattern-matching constructs.

Choosing the right flow control mechanism in modern C++20/23 ensures code is clean, readable, and maintainable, while taking advantage of new language features like pattern matching for safer and more expressive branching.

4.2 Exception Handling: try-catch and noexcept

Modern C++20/23 emphasizes robust and predictable error handling. Exception handling using try-catch and the noexcept specifier are critical tools for writing clean, maintainable, and reliable C++ code.

4.2.1 Principles of Exception Handling in Modern C++

1. Use Exceptions for Exceptional Conditions
    - Avoid using exceptions for normal control flow.
    - Only throw exceptions when an operation cannot continue safely.
2. Prefer noexcept Where Failure Is Impossible
    - Declaring functions as noexcept allows the compiler to optimize and improves safety.
    - Functions that guarantee no exceptions, such as destructors or simple arithmetic operations, should be marked noexcept.
3. Catch Exceptions by Reference
    - Always catch exceptions as const & to avoid slicing and unnecessary copies.
    - Example: catch(const std::runtime_error& e).
4. Provide Meaningful Error Messages
    - Include clear, actionable information in exception messages to aid debugging.
5. RAII and Resource Safety

- Combine exception handling with Resource Acquisition Is Initialization (RAII) to ensure automatic resource cleanup.

6. Avoid Empty catch Blocks

- Swallowing exceptions silently creates hidden bugs and violates clean code principles.

4.2.2 Example: Bad vs Clean Exception Handling

Bad Code:

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

int getElement(const std::vector<int>& v, int index) {
    return v.at(index); // may throw std::out_of_range
}

int main() {
    std::vector<int> numbers{1, 2, 3};
    try {
        std::cout << getElement(numbers, 5) << "\n";
    } catch (...) {
        // silently ignore exception
    }
}
```

Issues:

- Catching all exceptions with catch(...) hides the root cause.
- No meaningful feedback is provided to the user or developer.

- Makes debugging and maintenance difficult.

Clean Code: Using Proper try-catch

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

int getElement(const std::vector<int>& numbers, int index) {
    if (index < 0 || index >= static_cast<int>(numbers.size())) {
        throw std::out_of_range("Index " + std::to_string(index) + " is out of range");
    }
    return numbers[index];
}

int main() {
    std::vector<int> numbers{1, 2, 3};
    try {
        int value = getElement(numbers, 5);
        std::cout << value << "\n";
    } catch (const std::out_of_range& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}
```

Improvements:

- Catching std::out_of_range by reference provides specific error handling.
- Clear error message helps debugging.
- Avoids swallowing exceptions silently.

Advanced Clean Code: Using noexcept for Functions That Cannot Fail

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

int square(int value) noexcept {
    return value * value; // cannot throw
}

int main() {
    int x = 5;
    std::cout << "Square of " << x << " is " << square(x) << "\n";
}
```

Advantages:

- Declaring square as noexcept communicates intent and guarantees safety.
- Allows compiler optimizations.
- Improves code readability and reliability.

4.2.3 Key Takeaways

1. Use exceptions only for exceptional conditions, not normal control flow.
2. Always catch exceptions by reference and provide meaningful messages.
3. Mark functions noexcept when they are guaranteed not to throw.
4. Avoid empty catch blocks to prevent hidden bugs.
5. Combine RAII with exception handling for automatic and safe resource management.
6. Using modern C++20/23 best practices ensures robust, clean, and maintainable error handling.

Proper exception handling using try-catch and noexcept improves code safety, maintainability, and clarity, aligning with clean code principles in modern C++20/23.

4.3 Avoiding Complexity and Writing Short, Clear Functions

In modern C++20/23, maintainable and readable code relies heavily on keeping functions short, focused, and clear. Complex functions with multiple responsibilities or deeply nested logic reduce readability, increase the risk of bugs, and make testing difficult.

4.3.1 Principles for Short and Clear Functions

1. Single Responsibility Principle (SRP)
    - Each function should do one thing and do it well.
    - Smaller, focused functions improve readability, testability, and maintainability.
2. Limit Function Length
    - Ideally, functions should fit within 20–30 lines, though the focus is on clarity rather than strict line count.
    - Long functions indicate potential for refactoring into smaller units.
3. Minimize Nested Control Structures
    - Deeply nested if-else or loops reduce readability.
    - Use early returns or guard clauses to simplify flow.
4. Descriptive Naming for Actions
    - Short functions require clear names that describe exactly what they do.

- Combining expressive names with short functions improves self-documenting code.

5. Leverage Modern C++20/23 Features

- Use ranges, algorithms, lambdas, and structured bindings to reduce boilerplate loops and conditional complexity.
- This promotes concise, readable, and safe code.

4.3.2 Example: Bad vs Clean Code

Bad Code: Long, complex function

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

void processNumbers(const std::vector<int>& numbers) {
    for(int i = 0; i < static_cast<int>(numbers.size()); ++i) {
        if(numbers[i] % 2 == 0) {
            std::cout << numbers[i] << " is even\n";
        } else {
            if(numbers[i] > 0) {
                std::cout << numbers[i] << " is positive odd\n";
            } else {
                std::cout << numbers[i] << " is negative odd\n";
            }
        }
    }
}

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

```
    processNumbers(nums);
}
```

Issues:

- Single function does multiple responsibilities: checks parity, checks sign, prints messages.
- Nested if-else reduces readability.
- Hard to test or extend without introducing bugs.

Clean Code: Short, focused functions

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

bool isEven(int number) noexcept {
    return number % 2 == 0;
}

bool isPositive(int number) noexcept {
    return number > 0;
}

void printNumberInfo(int number) {
    if (isEven(number)) {
        std::cout << number << " is even\n";
    } else if (isPositive(number)) {
        std::cout << number << " is positive odd\n";
    } else {
        std::cout << number << " is negative odd\n";
    }
}
```

```
void processNumbers(const std::vector<int>& numbers) {
    for (int number : numbers) {
        printNumberInfo(number);
    }
}

int main() {
    std::vector<int> nums{1, 2, -3, 4, -5};
    processNumbers(nums);
}
```

Improvements:

- Single responsibility: isEven, isPositive, printNumberInfo.
- Simplified processNumbers with clear, high-level logic.
- Easier to test, extend, and maintain.
- Short, readable, and expressive functions aligned with clean code principles.

Advanced Modern C++20/23 Example: Using Ranges and Lambdas

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

void printEvenOdd(const std::vector<int>& numbers) {
    for (int number : numbers | std::views::transform([](int n) {
            if (n % 2 == 0) return "even";
            return n > 0 ? "positive odd" : "negative odd";
        }))
    {
```

```
        std::cout << number << "\n";
    }
}

int main() {
    std::vector<int> nums{1, 2, -3, 4, -5};
    printEvenOdd(nums);
}
```

Advantages:

- Uses C++20 ranges and lambdas for concise, readable processing.
- Reduces boilerplate loops and nested logic.
- Keeps high-level function responsibilities clear and minimal.

4.3.3 Key Takeaways

1. Functions should do one thing; multiple responsibilities indicate need for refactoring.
2. Limit nested control structures using early returns, guard clauses, or helper functions.
3. Short, descriptive functions improve readability, maintainability, and testability.
4. Modern C++20/23 features like ranges, lambdas, and structured bindings reduce complexity.
5. Following these principles ensures clean, maintainable, and modern program flow control.

Keeping functions short, clear, and focused is a cornerstone of clean code in modern C++20/23, reducing cognitive load and improving maintainability without sacrificing performance or expressiveness.

Chapter 5

Functions and Classes in Modern C++

5.1 Small Functions, Default Parameters, and Use of const and constexpr

In modern C++20/23, functions and classes should prioritize clarity, safety, and minimalism. Key strategies include writing small functions, leveraging default parameters, and using const and constexpr effectively. These principles increase readability, maintainability, and compile-time guarantees.

5.1.1 Principles

1. Small Functions

    - Each function should perform a single, well-defined task.
    - Improves readability, facilitates testing, and simplifies debugging.
    - Excessively long functions indicate opportunities for refactoring.

2. Default Parameters

    - Default parameters reduce boilerplate calls while keeping functions flexible.
    - Ensure defaults are meaningful and maintain clarity of intent.
    - Avoid overly complex default parameter logic that obscures function behavior.

3. const Correctness

    - Use const to indicate immutability of arguments, member functions, and variables.
    - Enhances readability and prevents unintended modifications.

4. constexpr for Compile-Time Computation

    - Functions and variables declared constexpr are evaluated at compile-time, improving performance and safety.
    - Modern C++20/23 extends constexpr to more complex logic, including loops, branching, and class member functions.

5.1.2 Example: Bad vs Clean Code

Bad Code: Long, mutable function without defaults

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

class Calculator {
public:
    int multiplyAdd(int a, int b, int c) {
        int result = a * b + c;
```

```
        return result;
    }
};

int main() {
    Calculator calc;
    std::cout << calc.multiplyAdd(2, 3, 5) << "\n";
}
```

Issues:

- Function is not const and cannot be safely called on const objects.
- Arguments are mutable, no compile-time guarantees.
- No default parameters; caller must provide all arguments.
- Single function does multiple responsibilities if expanded, e.g., error checking.

Clean Code: Small, const/constexpr-aware, default parameters

```
#include <iostream>

class Calculator {
public:
    constexpr int multiplyAdd(int a, int b, int c = 0) const noexcept {
        return a * b + c;
    }
};

int main() {
    constexpr Calculator calc{};
    constexpr int result = calc.multiplyAdd(2, 3); // uses default parameter
```

```
    std::cout << "Result: " << result << "\n";
}
```

Improvements:

- multiplyAdd is small and focused, performing a single arithmetic operation.
- Marked const: can be called on const objects.
- Marked constexpr: evaluated at compile-time for constant inputs.
- Added default parameter c = 0 for flexibility.
- noexcept communicates no exception guarantees.

Advanced Example: Using Modern C++20/23 Features

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

class Statistics {
public:
    template <size_t N>
    constexpr double mean(const std::array<int, N>& values, int scale = 1) const noexcept {
        int sum = 0;
        for (int value : values) sum += value;
        return static_cast<double>(sum) / N * scale;
    }
};

int main() {
    constexpr Statistics stats{};
    constexpr std::array<int, 5> numbers{1, 2, 3, 4, 5};
```

```
    constexpr double average = stats.mean(numbers); // default scale
    std::cout << "Average: " << average << "\n";
}
```

Advantages:

- Function is small, focused, and readable.
- template allows compile-time size checking.
- constexpr ensures compile-time evaluation for constant data.
- Default parameter scale adds flexibility without complicating calls.
- Combines modern C++20/23 capabilities with clean code principles.

5.1.3 Key Takeaways

1. Small, single-responsibility functions increase readability and maintainability.
2. Default parameters provide flexibility while keeping function calls concise.
3. Use const to guarantee immutability and improve safety.
4. Use constexpr to compute values at compile-time and enable optimization.
5. Combining these principles with C++20/23 features like templates and ranges produces robust, clear, and maintainable code.

Adhering to these practices ensures that functions and classes in modern C++20/23 are concise, predictable, and easy to understand, enabling cleaner, safer, and more maintainable software.

5.2 Designing Clean Classes (Classes/Structs)

In modern C++20/23, well-designed classes and structs are fundamental to clean, maintainable, and extensible code. Clean design emphasizes single responsibility, encapsulation, clear ownership semantics, and consistent naming, while leveraging modern C++ features for safety and expressiveness.

5.2.1 Principles for Clean Class Design

1. Single Responsibility Principle (SRP)
    - Each class should have one clear purpose.
    - Avoid mixing unrelated responsibilities (e.g., computation and I/O in the same class).
2. Encapsulation
    - Keep data private and expose behavior through public member functions.
    - Protect class invariants and prevent external misuse.
3. Constructors and Member Initialization
    - Use explicit constructors to prevent unintended conversions.
    - Prefer member initializer lists to assign values efficiently.
    - Use =default for trivial constructors or =delete to prevent unwanted operations.
4. const Correctness and noexcept
    - Mark read-only member functions as const.

- Declare functions as noexcept when guaranteed not to throw.

5. Struct vs Class

- Use structs for simple data aggregates without behavior.
- Use classes for types that encapsulate state and behavior.

6. Modern C++20/23 Features

- Consider [[nodiscard]] for functions whose results must not be ignored.
- Use constexpr members for compile-time constants.
- Leverage default, delete, and smart pointers for safe resource management.

5.2.2 Example: Bad vs Clean Class Design

Bad Code: A class mixing responsibilities and poor encapsulation

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

class DataManager {
public:
    std::vector<int> data;

    void add(int value) {
        data.push_back(value);
    }

    void printAll() {
        for(int i : data) std::cout << i << "\n";
    }
```

```
    int sum() {
        int s = 0;
        for(int i : data) s += i;
        return s;
    }
};

int main() {
    DataManager dm;
    dm.add(10);
    dm.add(20);
    dm.printAll();
    std::cout << "Sum: " << dm.sum() << "\n";
}
```

Issues:

- Public data member data violates encapsulation.
- Mixes responsibilities: storing data, printing, and computing sum.
- Functions not marked const or noexcept.
- Hard to maintain or extend safely.

Clean Code: Properly Designed Class

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

class DataManager {
private:
    std::vector<int> data_;
```

```
public:
    // Explicit default constructor
    DataManager() = default;

    // Deleted copy assignment to prevent accidental shallow copies
    DataManager(const DataManager&) = default;
    DataManager& operator=(const DataManager&) = delete;

    // Add value to data
    void add(int value) noexcept {
        data_.push_back(value);
    }

    // Compute sum of data
    [[nodiscard]] int sum() const noexcept {
        return std::accumulate(data_.begin(), data_.end(), 0);
    }

    // Print all values
    void printAll() const noexcept {
        for (int value : data_) {
            std::cout << value << "\n";
        }
    }
};

int main() {
    DataManager dm;
    dm.add(10);
    dm.add(20);

    dm.printAll();
```

```
    std::cout << "Sum: " << dm.sum() << "\n";
}
```

Improvements:

- Private data member data_ ensures encapsulation.
- Single responsibility per method: adding, summing, printing.
- Functions are const/noexcept where applicable.
- [[nodiscard]] warns if sum is ignored.
- Clean, maintainable, and aligned with modern C++20/23 practices.

Advanced Modern C++20/23 Example: Using constexpr and Aggregates

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

struct Stats {
    std::array<int, 5> values{};
    constexpr int sum() const noexcept {
        int total = 0;
        for (int v : values) total += v;
        return total;
    }
};

int main() {
    constexpr Stats stats{{1, 2, 3, 4, 5}};
    constexpr int total = stats.sum();
    std::cout << "Total: " << total << "\n";
}
```

Advantages:

- Stats as a struct for a simple data aggregate.
- constexpr allows compile-time evaluation.
- Maintains clarity, safety, and modern C++ expressiveness.

5.2.3 Key Takeaways

1. Encapsulate data and expose clear, focused behavior.
2. Single Responsibility Principle: each class should do one thing well.
3. Use const, noexcept, constexpr, =default, and =delete for safety and clarity.
4. Choose structs for simple aggregates, classes for complex types with behavior.
5. Modern C++20/23 features allow more expressive, safer, and maintainable classes.

Designing clean classes and structs in C++20/23 ensures your code is robust, readable, and maintainable, while fully leveraging modern language features for clarity, safety, and compile-time guarantees.

5.3 Core Principles: RAII, Rule of Five, Rule of Zero

Modern C++20/23 emphasizes safe resource management and predictable object behavior through core principles such as RAII (Resource Acquisition Is Initialization), Rule of Five, and Rule of Zero. Mastering these principles is essential for writing clean, maintainable, and exception-safe code.

5.3.1 RAII (Resource Acquisition Is Initialization)

Principle:

- Acquire resources in constructors and release them in destructors.
- Ensures resources are automatically released when objects go out of scope, eliminating memory leaks or dangling handles.
- Works seamlessly with exceptions, providing strong safety guarantees.

Bad Code: Manual Resource Management

```
#include <iostream>

void process() {
    int* data = new int[5];
    for(int i = 0; i < 5; ++i) data[i] = i;

    // exception occurs
    throw std::runtime_error("Error!");

    delete[] data; // never reached
}
```

```
int main() {
    try {
        process();
    } catch(...) {
        std::cout << "Exception caught\n";
    }
}
```

Issues:

- Memory leak occurs if exception is thrown before delete[].
- Manual cleanup is error-prone.

Clean Code: RAII using std::vector

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

void process() {
    std::vector<int> data(5);
    for(int i = 0; i < 5; ++i) data[i] = i;

    throw std::runtime_error("Error!"); // safe: no memory leak
}

int main() {
    try {
        process();
    } catch(const std::exception& e) {
        std::cout << e.what() << "\n";
    }
}
```

Improvements:

- std::vector automatically releases memory when going out of scope.
- Exception-safe, simpler, and more maintainable.

5.3.2 Rule of Five

Principle:

- If a class manages resources manually, you must explicitly define five special member functions:
    1. Destructor
    2. Copy Constructor
    3. Copy Assignment Operator
    4. Move Constructor
    5. Move Assignment Operator

Bad Code: Missing Copy/Move Safety

```
#include <iostream>

class Buffer {
    int* data_;
public:
    Buffer(size_t size) { data_ = new int[size]; }
    ~Buffer() { delete[] data_; }
};

int main() {
```

```
    Buffer b1(5);
    Buffer b2 = b1; // shallow copy, double delete!
}
```

Issues:

- Shallow copy leads to double deletion and undefined behavior.
- Manual resource management without defining copy/move operations is unsafe.

Clean Code: Proper Rule of Five

```
#include <iostream>
#include <algorithm>

class Buffer {
private:
    int* data_;
    size_t size_;
public:
    explicit Buffer(size_t size) : data_(new int[size]), size_(size) {}

    // Destructor
    ~Buffer() { delete[] data_; }

    // Copy Constructor
    Buffer(const Buffer& other) : data_(new int[other.size_]), size_(other.size_) {
        std::copy(other.data_, other.data_ + size_, data_);
    }

    // Copy Assignment
    Buffer& operator=(const Buffer& other) {
        if(this != &other) {
            delete[] data_;
```

```
            size_ = other.size_;
            data_ = new int[size_];
            std::copy(other.data_, other.data_ + size_, data_);
        }
        return *this;
    }

    // Move Constructor
    Buffer(Buffer&& other) noexcept : data_(other.data_), size_(other.size_) {
        other.data_ = nullptr;
        other.size_ = 0;
    }

    // Move Assignment
    Buffer& operator=(Buffer&& other) noexcept {
        if(this != &other) {
            delete[] data_;
            data_ = other.data_;
            size_ = other.size_;
            other.data_ = nullptr;
            other.size_ = 0;
        }
        return *this;
    }
};
```

Improvements:

- Safe copy and move semantics prevent resource leaks and double deletion.
- Follows modern C++20/23 best practices for manual resource management.

5.3.3 Rule of Zero

Principle:

- Prefer using RAII-compliant standard containers and smart pointers.
- Avoid defining any special member functions manually if possible.
- Reduces boilerplate and ensures automatic resource safety.

Clean Code: Rule of Zero Example

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

class Buffer {
private:
    std::vector<int> data_; // RAII
    std::unique_ptr<int[]> extra_; // RAII

public:
    explicit Buffer(size_t size) : data_(size), extra_(std::make_unique<int[]>(size)) {}

    void fill() {
        for(size_t i = 0; i < data_.size(); ++i) {
            data_[i] = static_cast<int>(i);
            extra_[i] = static_cast<int>(i * 2);
        }
    }

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

```
    }
};

int main() {
    Buffer buf(5);
    buf.fill();
    buf.print();
}
```

Advantages:

- No need to manually define destructor, copy, or move constructors.
- Safe, maintainable, and fully leverages modern C++20/23 RAII.
- Clean and concise, adhering to Rule of Zero.

5.3.4 Key Takeaways

1. RAII ensures automatic, exception-safe resource management.
2. Rule of Five: implement all five special member functions when managing resources manually.
3. Rule of Zero: prefer RAII-compliant containers and smart pointers to avoid boilerplate.
4. Modern C++20/23 features like std::unique_ptr, std::vector, and constexpr allow safer, cleaner, and maintainable designs.
5. Following these principles prevents memory leaks, undefined behavior, and improves code clarity and maintainability.

Applying RAII, Rule of Five, and Rule of Zero is essential for writing robust, modern, and clean C++20/23 classes, ensuring safe resource handling and reducing maintenance overhead.

5.4 Proper Usage of std::unique_ptr and std::shared_ptr

Modern C++20/23 promotes safe, automatic memory management through smart pointers. Using std::unique_ptr and std::shared_ptr correctly reduces the risk of memory leaks, dangling pointers, and undefined behavior. Proper understanding of ownership semantics and lifecycle management is essential for clean, maintainable code.

5.4.1 Principles

1. std::unique_ptr
    - Provides exclusive ownership of a resource.
    - Automatically deletes the resource when the unique_ptr goes out of scope.
    - Cannot be copied, but can be moved.
    - Preferred when single ownership is sufficient, improving clarity and safety.
2. std::shared_ptr
    - Provides shared ownership of a resource.
    - Deletes the resource when the last shared pointer goes out of scope.
    - Use std::weak_ptr to avoid reference cycles.
    - Use cautiously to prevent unintentional performance overhead.
3. Avoid raw new/delete
    - Smart pointers encapsulate resource management and eliminate manual cleanup errors.
    - They improve exception safety by ensuring proper cleanup on scope exit.

4. Prefer std::make_unique and std::make_shared

    - Safer and more efficient than directly calling new.
    - Avoids resource leaks if exceptions occur during construction.

5.4.2 Example: Bad vs Clean Code

Bad Code: Manual memory management

```
#include <iostream>

class Widget {
public:
    void sayHi() { std::cout << "Hello from Widget\n"; }
};

int main() {
    Widget* w = new Widget();
    w->sayHi();

    // forget to delete -> memory leak
    // delete w;
}
```

Issues:

- Raw pointer allocation is error-prone.
- Manual deletion is forgotten, causing memory leaks.
- Exception safety not guaranteed.

Clean Code: Using std::unique_ptr

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

class Widget {
public:
    void sayHi() const { std::cout << "Hello from Widget\n"; }
};

int main() {
    auto w = std::make_unique<Widget>(); // unique ownership
    w->sayHi();
} // automatic deletion when `w` goes out of scope
```

Improvements:

- No need for manual delete.
- Ownership is clear: only one unique_ptr owns the resource.
- Exception-safe: resource released automatically on scope exit.

Clean Code: Using std::shared_ptr for shared ownership

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

class Widget {
public:
    void sayHi() const { std::cout << "Hello from Widget\n"; }
};

void greetWidget(std::shared_ptr<Widget> w) {
    w->sayHi();
}
```

```
int main() {
    auto w = std::make_shared<Widget>(); // shared ownership
    greetWidget(w);
    greetWidget(w); // safe shared usage
} // Widget deleted automatically when last shared_ptr goes out of scope
```

Advantages:

- Shared ownership allows multiple parts of the program to safely use the same resource.
- Automatic cleanup prevents leaks.
- Avoids copying raw pointers and manual cleanup errors.

5.4.3 Advanced Modern C++20/23 Practices

1. Use const with smart pointers where appropriate

```
const auto w = std::make_unique<Widget>();
w->sayHi();
```

1. Prefer passing std::shared_ptr by value if ownership is shared, or by const reference if just observing.

```
void observeWidget(const std::shared_ptr<Widget>& w) {
    w->sayHi();
}
```

1. Avoid mixing unique_ptr and shared_ptr for the same resource, which can lead to undefined behavior.
2. Use weak_ptr to break cycles and prevent memory leaks with shared_ptr.

```cpp
#include <memory>

struct Node {
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev; // avoids cycle
};
```

5.4.4 Key Takeaways

1. Prefer unique_ptr for exclusive ownership and simplicity.
2. Use shared_ptr only when multiple owners are necessary, and manage cycles with weak_ptr.
3. Always prefer std::make_unique and std::make_shared to construct smart pointers safely.
4. Smart pointers ensure exception safety, predictable destruction, and maintainable ownership semantics.
5. Proper use of smart pointers aligns with modern C++20/23 clean code principles and eliminates manual memory errors.

Correct use of std::unique_ptr and std::shared_ptr ensures safe, maintainable, and modern C++ code, freeing developers from manual memory management while maintaining clarity and expressiveness.

Chapter 6

Templates and Generic Programming

6.1 Writing Safe, Clean Templates

Templates are a cornerstone of modern C++20/23, enabling generic, reusable, and type-safe code. However, improper use can lead to hard-to-read, error-prone, and unmaintainable code. Writing clean templates requires clarity, constraints, and proper separation of responsibilities.

6.1.1 Principles of Clean Template Design

1. Single Responsibility

    - Templates should focus on one generic task, avoiding mixing multiple unrelated behaviors.

2. Use Concepts and Constraints (C++20/23)

    - Introduce concepts or requires clauses to enforce type safety at compile-time.

- Improves readability and prevents misuse with unsupported types.

3. Prefer constexpr for Compile-Time Evaluation
   - When possible, leverage compile-time computation for templates.
4. Avoid Excessive Template Specialization
   - Specializations can obscure behavior; prefer if constexpr or concepts.
5. Readable Naming and Parameter Types
   - Template parameters should clearly indicate intended type or role.
   - Avoid vague names like T1, T2 unless context is obvious.
6. Separate Implementation
   - For larger templates, separate declaration and definition clearly for readability.
   - Consider inline functions in headers for small templates.

6.1.2 Example: Bad vs Clean Template

Bad Code: Unsafe and unclear template

```
#include <iostream>

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

```
int main() {
    std::cout << add(1, 2) << "\n";       // ok
    std::cout << add(1, 2.5) << "\n";     // implicit conversion, unexpected behavior
}
```

Issues:

- No type constraints: mixed types can cause implicit conversions or errors.
- Lacks clarity on supported operations.
- Potentially unsafe for complex types that do not support +.

Clean Code: Using Concepts and constexpr

```
#include <concepts>
#include <iostream>

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

int main() {
    std::cout << add(1, 2) << "\n";       // ok
    // std::cout << add(1, 2.5);          // compile-time error
}
```

Improvements:

- std::integral concept restricts template to integral types.
- constexpr allows compile-time evaluation.
- noexcept clarifies no exceptions will be thrown.

- Prevents misuse with floating-point or non-addable types.

Advanced Example: Using C++20 Concepts and if constexpr

```
#include <concepts>
#include <iostream>

template <typename T>
constexpr T multiplyAdd(T a, T b, T c) noexcept {
    if constexpr (std::is_integral_v<T>) {
        return a * b + c;
    } else {
        return static_cast<T>(a * b) + c; // for floating-point types
    }
}

int main() {
    std::cout << multiplyAdd(2, 3, 4) << "\n";      // integer
    std::cout << multiplyAdd(2.0, 3.5, 4.0) << "\n"; // floating-point
}
```

Advantages:

- Uses if constexpr for conditional behavior based on type.
- Supports multiple type categories safely.
- Maintains compile-time clarity and safety.
- Ensures templates are predictable and clean while remaining generic.

6.1.3 Key Takeaways

1. Templates should be focused, single-purpose, and predictable.

2. Use concepts and requires clauses to enforce constraints and improve readability.
3. Prefer constexpr and noexcept for safe, compile-time capable templates.
4. Avoid implicit conversions or excessive specialization, which reduce clarity.
5. Clear naming of template parameters improves maintainability and comprehension.

Writing safe, clean templates ensures that your generic code is reusable, predictable, and maintainable, leveraging C++20/23 features for strong compile-time type safety and clear code semantics.

6.2 Using Concepts (C++20) to Improve Clarity of Signatures

C++20 introduced concepts, a powerful tool to express template requirements directly in the function or class signature. Concepts improve readability, maintainability, and compile-time type safety by explicitly constraining template parameters, making your templates easier to understand and use correctly.

6.2.1 Principles for Using Concepts

1. Express Intended Behavior in the Signature
    - Use concepts to communicate what types a template accepts, instead of relying on implicit operations or assumptions.
2. Replace SFINAE with Clear Constraints
    - Concepts provide a cleaner, more readable alternative to enable_if and complex SFINAE.
3. Use Standard Library Concepts
    - C++20 provides predefined concepts like std::integral, std::floating_point, std::totally_ordered, std::ranges::range, etc.
    - Leverage them for consistency and clarity.
4. Custom Concepts for Domain-Specific Constraints
    - Define your own concepts when needed to enforce specific properties of template parameters.
5. Enhance Error Messages

- Concepts produce clearer compiler errors, indicating exactly which requirement failed, improving developer experience.

6.2.2 Example: Bad vs Clean Template Signatures

Bad Code: Ambiguous Template Without Constraints

```
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add(1, 2) << "\n";       // ok
    std::cout << add(1, 2.5) << "\n";     // compiles but may be unexpected
}
```

Issues:

- Implicit conversions are allowed, potentially causing unexpected behavior.
- Template signature does not clearly express acceptable types.
- Error messages are hard to understand if a non-addable type is passed.

Clean Code: Using Standard Concepts

```
#include <concepts>
#include <iostream>

template <std::integral T>
T add(T a, T b) noexcept {
```

```
    return a + b;
}

int main() {
    std::cout << add(1, 2) << "\n";       // ok
    // std::cout << add(1, 2.5);          // compile-time error
}
```

Improvements:

- std::integral clearly restricts template to integer types.
- Signature is self-documenting, expressing the intent directly.
- Errors are immediate and clear if constraints are violated.

Advanced Example: Custom Concept for Addable Types

```
#include <concepts>
#include <iostream>

template <typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::same_as<T>;
};

template <Addable T>
T add(T a, T b) noexcept {
    return a + b;
}

struct NotAddable {};

int main() {
```

```
    std::cout << add(2, 3) << "\n";   // ok
    // NotAddable x, y;
    // std::cout << add(x, y);        // compile-time error
}
```

Advantages:

- Custom concept defines semantic constraints on type usage.
- Template clearly communicates its requirements, enhancing readability.
- Prevents misuse of the template with incompatible types.

6.2.3 Key Takeaways

1. Concepts improve template readability and safety by expressing constraints in the signature.
2. Replace SFINAE or enable_if hacks with clear, modern C++20 syntax.
3. Use standard concepts when possible and custom concepts for domain-specific constraints.
4. Concepts produce better compiler diagnostics, making code more maintainable and easier to debug.
5. Clean template signatures reduce unexpected behavior and implicit conversions, enhancing safety in modern C++20/23 code.

Using concepts in C++20 transforms templates from ambiguous and error-prone into clear, safe, and self-documenting constructs, aligning with clean code principles and ensuring robust, maintainable generic programming.

6.3 Writing Reusable and Extensible Code

Modern C++20/23 emphasizes writing code that is reusable, maintainable, and extensible. Templates and generic programming are central to achieving these goals, enabling type-independent operations while preserving safety and clarity.

6.3.1 Principles for Reusable and Extensible Code

1. Single Responsibility and Modularity
    - Each template or function should focus on a single operation.
    - Avoid tightly coupling multiple unrelated behaviors.
2. Use Concepts and Constraints
    - Constrain templates with concepts to ensure type safety and clear expectations.
    - Improves readability and prevents misuses.
3. Leverage Generic Algorithms and Ranges
    - Favor standard library algorithms and ranges over writing specialized loops.
    - Enhances reusability and reduces boilerplate.
4. Minimize Hardcoding Types or Behaviors
    - Use templates or type-erasure techniques to handle different types generically.
    - This allows extending functionality without rewriting code.
5. Keep Interfaces Stable and Predictable

- Reusable components should not expose internal implementation details.
- Provide clean and minimal public interfaces.

6.3.2 Example: Bad vs Clean Code

Bad Code: Non-Reusable and Hardcoded

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

int sumIntVector(const std::vector<int>& v) {
    int sum = 0;
    for(int x : v) sum += x;
    return sum;
}

int main() {
    std::vector<int> vec = {1, 2, 3};
    std::cout << sumIntVector(vec) << "\n";

    // Need to sum doubles? Must rewrite function
}
```

Issues:

- Hardcoded for std::vector<int> only.
- Not reusable for other containers or types.
- Extending requires writing new specialized functions, violating DRY principle.

Clean Code: Reusable Template with Concepts

```cpp
#include <concepts>
#include <ranges>
#include <numeric>
#include <iostream>

template <std::ranges::range R>
requires std::integral<std::ranges::range_value_t<R>> ||
         std::floating_point<std::ranges::range_value_t<R>>
auto sumRange(const R& container) {
    using T = std::ranges::range_value_t<R>;
    return std::accumulate(container.begin(), container.end(), T{0});
}

int main() {
    std::vector<int> vecInt = {1, 2, 3};
    std::vector<double> vecDouble = {1.1, 2.2, 3.3};

    std::cout << sumRange(vecInt) << "\n";       // 6
    std::cout << sumRange(vecDouble) << "\n";    // 6.6
}
```

Improvements:

- Works with any range of integral or floating-point types.
- Single, reusable function instead of multiple type-specific implementations.
- Explicit concept constraints clarify acceptable types.
- Extensible: works with vectors, arrays, or custom ranges.

6.3.3 Advanced Example: Extensible Generic Algorithm

```cpp
#include <concepts>
```

```cpp
#include <ranges>
#include <numeric>
#include <iostream>
#include <vector>
#include <list>

template <std::ranges::range R, typename Op>
auto accumulateRange(const R& container, Op op) {
    using T = std::ranges::range_value_t<R>;
    T result{};
    for (const auto& val : container) {
        result = op(result, val);
    }
    return result;
}

int main() {
    std::vector<int> v = {1, 2, 3};
    std::list<double> l = {1.5, 2.5, 3.5};

    auto sumInts = accumulateRange(v, [](auto a, auto b){ return a + b; });
    auto prodDoubles = accumulateRange(l, [](auto a, auto b){ return a * b; });

    std::cout << sumInts << "\n";       // 6
    std::cout << prodDoubles << "\n";   // 13.125
}
```

Advantages:

- Fully generic: works with any range and custom operation.
- Encourages extensibility: new operations can be defined without changing the template.

- Clean interface and safe, predictable behavior for any compatible type.

6.3.4 Key Takeaways

1. Reusable code should avoid hardcoded types and behaviors, relying on templates and generic programming.
2. Concepts clearly express constraints and expectations, improving readability.
3. Leveraging standard algorithms and ranges reduces boilerplate and increases consistency.
4. Extensible code allows adding new behaviors without modifying existing templates, following open/closed principle.
5. Properly written generic code in C++20/23 ensures clean, maintainable, and safe reuse across a wide range of types and containers.

Writing reusable and extensible templates ensures that your C++20/23 code is adaptable, maintainable, and robust, reducing duplication while maintaining clarity and safety.

Chapter 7

Modern Programming with STL and Ranges

7.1 Effective Use of STL Containers

The Standard Template Library (STL) provides a collection of highly optimized, type-safe containers in C++20/23. Using STL containers effectively enhances readability, maintainability, and safety, while reducing the need for custom data structures and manual memory management.

7.1.1 Principles for Effective Use of STL Containers

1. Choose the Right Container for the Job

    - std::vector: contiguous memory, best for dynamic arrays with frequent access by index.
    - std::list / std::forward_list: efficient insertions/deletions but slow random access.
    - std::deque: double-ended queue, good for frequent front and back insertions.

- std::map / std::unordered_map: key-value associations, ordered vs hashed.
- std::set / std::unordered_set: unique elements, ordered vs hashed.

2. Prefer STL Over Custom Containers

   - STL containers are well-tested, exception-safe, and maintainable.
   - Avoid reimplementing standard structures unless there is a compelling reason.

3. Use emplace over insert when possible

   - Reduces unnecessary copies and improves efficiency.

4. Leverage Type Safety and Iterators

   - Iterators provide generic access without exposing implementation details.
   - Range-based loops and STL algorithms improve clarity.

5. Avoid Manual Memory Management

   - Containers manage memory automatically, preventing leaks and dangling pointers.

7.1.2 Example: Bad vs Clean Code

Bad Code: Manual array management

```
#include <iostream>

int main() {
    int* arr = new int[5]{1, 2, 3, 4, 5};
    int sum = 0;
```

```
    for(int i = 0; i < 5; ++i) {
        sum += arr[i];
    }
    std::cout << "Sum: " << sum << "\n";

    // forget to delete -> memory leak
    // delete[] arr;
}
```

Issues:

- Manual memory allocation is error-prone.
- Hardcoded size limits flexibility.
- No exception safety; delete[] might be skipped if an exception occurs.

Clean Code: Using std::vector

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

int main() {
    std::vector<int> vec{1, 2, 3, 4, 5};
    int sum = std::accumulate(vec.begin(), vec.end(), 0);
    std::cout << "Sum: " << sum << "\n";
}
```

Improvements:

- Memory is automatically managed.
- Vector size is dynamic; can grow or shrink.

- Code is concise, readable, and exception-safe.

Advanced Example: Using STL with Modern C++20/23 Features

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

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

    // Use ranges for transformation and filtering
    auto evenSquares = vec | std::views::filter([](int x){ return x % 2 == 0; })
                           | std::views::transform([](int x){ return x * x; });

    for (int x : evenSquares) {
        std::cout << x << " "; // 4 16
    }
    std::cout << "\n";
}
```

Advantages:

- Readable, declarative style using ranges and views.
- No temporary containers needed.
- Works efficiently for large datasets, and integrates seamlessly with STL containers.

7.1.3 Key Takeaways

1. Choose the right container based on access patterns and performance requirements.

2. Prefer STL containers over raw arrays or custom implementations.
3. Use emplace and STL algorithms to reduce boilerplate and improve performance.
4. Leverage iterators, range-based loops, and C++20/23 ranges for clarity and maintainability.
5. Proper use of STL containers aligns with clean code principles and modern C++ best practices.

Effective use of STL containers allows developers to write safe, maintainable, and highly expressive C++20/23 code, freeing them from manual memory management while improving clarity and performance.

7.2 Ranges and Views in C++20

C++20 introduced Ranges and Views, revolutionizing how we work with STL containers and sequences. They allow declarative, composable, and lazy operations over collections, improving readability, efficiency, and maintainability.

7.2.1 Principles of Using Ranges and Views

1. Declarative Operations
    - Use ranges to express what to do with a sequence, rather than manually iterating with loops.
2. Lazy Evaluation with Views
    - Views are non-owning, lazy adapters. They do not create intermediate containers, improving performance.
3. Composable Pipelines
    - Combine multiple transformations (filtering, mapping, slicing) using | syntax for clean, readable pipelines.
4. Separation of Data and Operations
    - Ranges promote functional style by separating data storage (containers) from data processing (views/algorithms).
5. Safety and Clarity
    - Ranges eliminate manual indexing, reducing off-by-one errors and improving readability.

7.2.2 Example: Bad vs Clean Code

Bad Code: Manual Loop Processing

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

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

    for(size_t i = 0; i < vec.size(); ++i) {
        if (vec[i] % 2 == 0) {
            evenSquares.push_back(vec[i] * vec[i]);
        }
    }

    for(size_t i = 0; i < evenSquares.size(); ++i) {
        std::cout << evenSquares[i] << " "; // 4 16
    }
    std::cout << "\n";
}
```

Issues:

- Manual indexing increases risk of off-by-one errors.
- Temporary container is created manually, adding boilerplate.
- Hard to read and maintain, especially when operations become more complex.

Clean Code: Using Ranges and Views (C++20)

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

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

    auto evenSquares = vec
        | std::views::filter([](int x){ return x % 2 == 0; })
        | std::views::transform([](int x){ return x * x; });

    for (int x : evenSquares) {
        std::cout << x << " "; // 4 16
    }
    std::cout << "\n";
}
```

Improvements:

- Declarative style expresses what is being done, not how.
- No manual indexing; safer and more readable.
- Lazy evaluation: no temporary container is created; only iterated when needed.
- Easily extendable by adding additional view adaptors.

Advanced Example: Composable Pipeline

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

int main() {
```

```
    std::vector<int> numbers{1,2,3,4,5,6,7,8,9,10};

    auto result = numbers
        | std::views::filter([](int x){ return x % 3 == 0; })
        | std::views::transform([](int x){ return x * x; })
        | std::views::take(2);

    for (int x : result) {
        std::cout << x << " "; // 9 36
    }
    std::cout << "\n";
}
```

Advantages:

- Supports complex data transformations in a single, readable expression.
- Composable, maintainable, and extendable.
- Enhances clarity, safety, and performance by avoiding unnecessary copies.

7.2.3 Key Takeaways

1. Ranges and views make code declarative, safe, and readable.
2. Lazy evaluation avoids unnecessary temporary containers, improving performance.
3. Composable pipelines allow multiple transformations to be expressed concisely.
4. Eliminates manual iteration and indexing, reducing bugs.
5. Ranges integrate seamlessly with STL containers, providing a modern and clean approach to sequence processing in C++20/23.

Using ranges and views effectively ensures that your C++20/23 code is clean, expressive, and maintainable, allowing complex operations to be written safely and concisely while adhering to modern best practices.

7.3 Writing Clean Loops — Traditional vs Modern Styles

Writing clean loops is a cornerstone of readable, maintainable C++ code. Modern C++20/23 provides range-based loops, STL algorithms, and ranges to replace verbose traditional loops, reducing boilerplate, increasing safety, and clarifying intent.

7.3.1 Principles for Clean Loops

1. Prefer Range-Based Loops over Index-Based Loops
    - Avoid manual indexing whenever possible to reduce off-by-one errors.
2. Use STL Algorithms for Declarative Behavior
    - Replace explicit loops with algorithms like std::for_each, std::transform, std::accumulate for clarity and intent.
3. Leverage Ranges and Views for Filtering and Transformation
    - Compose operations without intermediate containers, maintaining laziness and efficiency.
4. Minimize Side Effects
    - Keep loops focused; avoid modifying unrelated state inside loops to enhance maintainability and testability.
5. Make Loops Readable and Self-Documenting
    - Use descriptive variable names and clear loop boundaries.

7.3.2 Example: Bad vs Clean Loops

Bad Code: Traditional Index-Based Loop

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

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

    for (size_t i = 0; i < numbers.size(); ++i) {
        if (numbers[i] % 2 == 0) {
            squares.push_back(numbers[i] * numbers[i]);
        }
    }

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

Issues:

- Verbose and error-prone due to manual indexing.
- Hard to read; the intent (filter even numbers and square them) is obscured.
- Temporary container created manually, increasing boilerplate.

Clean Code: Range-Based Loop

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

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

    for (int n : numbers) {
        if (n % 2 == 0) {
            squares.push_back(n * n);
        }
    }

    for (int s : squares) {
        std::cout << s << " ";
    }
    std::cout << "\n";
}
```

Improvements:

- Eliminates manual indexing, reducing potential bugs.
- Cleaner, more readable, and self-documenting.
- Maintains clear separation of filtering and transformation logic.

Modern Clean Code: Using Ranges and Views (C++20)

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

int main() {
```

```
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    auto squares = numbers
        | std::views::filter([](int n){ return n % 2 == 0; })
        | std::views::transform([](int n){ return n * n; });

    for (int s : squares) {
        std::cout << s << " ";
    }
    std::cout << "\n";
}
```

Advantages:

- Fully declarative: expresses what to do instead of how.
- No temporary container is created; lazy evaluation improves efficiency.
- Clear intent and maintainability: filter and transform operations are explicit.
- Easy to extend by adding additional view adaptors like take, drop, or stride.

7.3.3 Key Takeaways

1. Prefer modern C++ loop constructs over traditional index-based loops.
2. Range-based loops improve readability and reduce errors.
3. STL algorithms and ranges allow declarative, maintainable, and efficient iteration.
4. Loops should clearly express intent, focusing on what needs to be done rather than how to iterate.

5. Modern C++20/23 enables clean, reusable, and composable loop operations that adhere to clean code principles.

Using modern C++ loops and ranges ensures code is safe, readable, and maintainable, turning potentially verbose and error-prone operations into concise, declarative, and robust solutions.

7.4 Examples of Functional-Style Clean Code

Functional programming techniques in C++20/23, when combined with STL algorithms, ranges, and views, enable developers to write clean, declarative, and maintainable code. Functional-style code emphasizes pure operations, immutability, and composability, reducing side effects and improving readability.

7.4.1 Principles of Functional-Style Clean Code

1. Immutability
    - Avoid modifying containers or variables in place unless necessary.
    - Favor returning new collections with transformations applied.
2. Pure Functions
    - Functions should produce the same output for the same input without modifying external state.
3. Declarative Operations
    - Express what needs to be done rather than how to iterate.
    - Replace explicit loops with STL algorithms or ranges pipelines.
4. Composition
    - Combine multiple operations into pipelines for clarity and conciseness.
5. Minimal Side Effects
    - Limit I/O or global state modifications inside functional pipelines.

7.4.2 Example: Bad vs Clean Functional-Style Code

Bad Code: Imperative Loop-Based Processing

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

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

    for (size_t i = 0; i < numbers.size(); ++i) {
        if (numbers[i] % 2 == 0) {
            doubledEvens.push_back(numbers[i] * 2);
        }
    }

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

Issues:

- Manual loops obscure intent.
- Temporary container manipulated in place.
- Harder to extend or modify pipeline of operations.

Clean Code: Functional-Style Using STL Algorithms

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

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

    std::copy_if(numbers.begin(), numbers.end(), std::back_inserter(doubledEvens),
                 [](int n){ return n % 2 == 0; });

    std::transform(doubledEvens.begin(), doubledEvens.end(), doubledEvens.begin(),
                   [](int n){ return n * 2; });

    for (int x : doubledEvens) {
        std::cout << x << " "; // 4 8
    }
    std::cout << "\n";
}
```

Improvements:

- Separate filtering and transformation into clear, reusable operations.
- Avoid manual indexing; code expresses what is being done.
- Easier to extend or reuse individual steps.

Modern Clean Code: Functional-Style Using Ranges and Views (C++20)

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

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

    auto doubledEvens = numbers
        | std::views::filter([](int n){ return n % 2 == 0; })
        | std::views::transform([](int n){ return n * 2; });

    for (int x : doubledEvens) {
        std::cout << x << " "; // 4 8
    }
    std::cout << "\n";
}
```

Advantages:

- Fully declarative: expresses the filter-transform pipeline without manual iteration.
- Lazy evaluation: no intermediate container is created.
- Easy to compose additional operations like take, drop, or stride.
- Promotes pure functional principles and clean, maintainable code.

7.4.3 Advanced Functional-Style Example

```
#include <iostream>
#include <vector>
#include <ranges>
#include <numeric>

int main() {
    std::vector<int> numbers = {1,2,3,4,5,6,7,8,9,10};

    auto sumOfSquaresOfOdds = std::accumulate(
```

```
    numbers
    | std::views::filter([](int n){ return n % 2 != 0; })
    | std::views::transform([](int n){ return n * n; })
    , 0
  );

  std::cout << "Sum of squares of odd numbers: " << sumOfSquaresOfOdds << "\n"; // 165
}
```

Advantages:

- Combines filtering, transformation, and reduction in a concise, readable pipeline.
- Promotes clean, functional-style operations without mutable intermediate state.
- Aligns with modern C++20 best practices for readability, maintainability, and safety.

7.4.4 Key Takeaways

1. Functional-style programming in C++20/23 improves clarity and maintainability.
2. Ranges and views enable lazy evaluation and composable operations.
3. Use pure functions and minimal side effects for clean pipelines.
4. Functional-style code expresses intent explicitly, reducing boilerplate and errors.
5. Combining STL algorithms, ranges, and views allows concise, declarative, and efficient solutions for modern C++ programming.

Functional-style programming using C++20/23 ranges and views ensures code is clean, safe, and extensible, turning loops and transformations into expressive pipelines that clearly convey intent while minimizing manual state management.

Chapter 8

Modern Memory Management

8.1 Using RAII and Smart Pointers

Proper memory management is a cornerstone of safe and maintainable C++ code. Modern C++ (C++20/23) emphasizes RAII (Resource Acquisition Is Initialization) and smart pointers (std::unique_ptr, std::shared_ptr, std::weak_ptr) to automate resource management, eliminate leaks, and improve exception safety.

8.1.1 Principles of RAII and Smart Pointers

1. RAII Concept

    - Tie the lifetime of resources (memory, files, sockets) to the lifetime of objects.
    - Resource acquisition happens in the constructor; release happens in the destructor.

2. Use Smart Pointers Instead of Raw Pointers

- std::unique_ptr for exclusive ownership.
- std::shared_ptr for shared ownership.
- std::weak_ptr to break cycles in shared ownership.

3. Avoid Manual new and delete
   - Manual memory management is error-prone and reduces maintainability.
4. Ensure Exception Safety
   - RAII guarantees automatic cleanup when exceptions occur, preventing leaks.
5. Prefer Stack Allocation When Possible
   - Stack objects are fast, simple, and automatically cleaned up. Use heap allocation only when necessary.

8.1.2 Example: Bad vs Clean Code

Bad Code: Manual Memory Management

```
#include <iostream>

class Widget {
public:
    Widget() { std::cout << "Widget created\n"; }
    ~Widget() { std::cout << "Widget destroyed\n"; }
};

int main() {
    Widget* w = new Widget();
```

```
    // some operations
    if (true) {
        std::cout << "Doing something...\n";
        // forgot to delete in case of early return or exception
    }

    delete w; // might be skipped if exception occurs
}
```

Issues:

- Manual new/delete is error-prone.
- Exception safety is not guaranteed.
- Harder to maintain in complex codebases.

Clean Code: Using RAII with Stack Allocation

```
#include <iostream>

class Widget {
public:
    Widget() { std::cout << "Widget created\n"; }
    ~Widget() { std::cout << "Widget destroyed\n"; }
};

int main() {
    Widget w; // RAII ensures automatic cleanup
    std::cout << "Doing something...\n";
} // Destructor automatically called
```

Improvements:

- No manual delete needed.
- Object automatically cleaned up at scope exit.
- Exception-safe and maintainable.

Modern Clean Code: Using std::unique_ptr (C++20/23)

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

class Widget {
public:
    Widget() { std::cout << "Widget created\n"; }
    ~Widget() { std::cout << "Widget destroyed\n"; }
};

int main() {
    auto w = std::make_unique<Widget>(); // unique_ptr ensures RAII

    std::cout << "Doing something with widget...\n";

    // w automatically deleted at scope exit
}
```

Advantages:

- Automatic cleanup: RAII guarantees destructor call.
- No risk of memory leaks even if exceptions occur.
- std::make_unique is exception-safe and clear.

Advanced Example: Using std::shared_ptr for Shared Ownership

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

class Widget {
public:
    Widget() { std::cout << "Widget created\n"; }
    ~Widget() { std::cout << "Widget destroyed\n"; }
};

int main() {
    std::shared_ptr<Widget> w1 = std::make_shared<Widget>();
    std::shared_ptr<Widget> w2 = w1; // shared ownership

    std::cout << "Both owners can use the widget\n";
} // Widget destroyed automatically when last owner goes out of scope
```

Advantages:

- Shared ownership handled safely.
- No need to manually track references.
- Exception-safe and clean.

8.1.3 Key Takeaways

1. RAII ensures automatic, exception-safe resource management.
2. Prefer smart pointers over raw pointers to manage heap resources.
3. Use std::unique_ptr for exclusive ownership, std::shared_ptr for shared ownership, and std::weak_ptr to break reference cycles.
4. Stack allocation is preferred when possible for simplicity and performance.

5. Proper memory management using RAII and smart pointers leads to clean, maintainable, and robust C++20/23 code.

Using RAII and smart pointers effectively ensures your C++ code is safe, exception-resilient, and maintainable, eliminating the common pitfalls of manual memory management while embracing modern C++20/23 best practices.

8.2 Avoiding Raw Pointers Whenever Possible

Raw pointers (T*) were once a fundamental tool in C++ for dynamic memory management. However, with the introduction and maturity of smart pointers, automatic lifetime tracking, and RAII (Resource Acquisition Is Initialization) principles in Modern C++ (C++11 and later), the use of raw pointers for ownership has become a liability rather than a necessity.
Modern C++ (C++20/23) discourages managing object lifetimes manually. Instead, developers should treat raw pointers as non-owning observers—used only to reference objects managed elsewhere.

8.2.1 The Problem with Raw Pointers

Raw pointers require explicit new and delete operations. This manual management leads to several common issues:

- Memory leaks when delete is forgotten.
- Dangling pointers when deleted objects are still referenced.
- Exception unsafety: if an exception occurs before delete, the resource is never released.
- Unclear ownership semantics — it's not obvious who "owns" the resource.

These problems make raw pointers one of the leading causes of unstable, insecure, and hard-to-maintain C++ code.

8.2.2 Modern C++ Philosophy: Ownership Must Be Explicit

Modern C++ mandates that every resource must have a clear owner:

- Use automatic storage (stack variables) whenever possible.
- Use std::unique_ptr for exclusive ownership.
- Use std::shared_ptr for shared ownership.
- Use raw pointers or references only for observation, not ownership.

This explicit ownership model simplifies reasoning about object lifetime, improves safety, and ensures exception resilience.

8.2.3 Example: Bad Code vs Clean Code

- Bad Code: Using Raw Pointers for Ownership

```
#include <iostream>

class Logger {
public:
    Logger() { std::cout << "Logger initialized\n"; }
    ~Logger() { std::cout << "Logger destroyed\n"; }

    void log(const std::string& msg) {
        std::cout << "[LOG] " << msg << "\n";
    }
};

void process() {
    Logger* logger = new Logger(); // manual ownership
    logger->log("Processing started...");

    if (true) {
        std::cout << "An error occurred!\n";
```

```
        return; // forgot to delete -> memory leak
    }

    delete logger;
}

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

Problems:

- The function process() may return early without releasing memory.
- If exceptions are thrown, cleanup never occurs.
- Ownership is unclear—who deletes the pointer and when?

- Clean Code: Using std::unique_ptr

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

class Logger {
public:
    Logger() { std::cout << "Logger initialized\n"; }
    ~Logger() { std::cout << "Logger destroyed\n"; }

    void log(const std::string& msg) {
        std::cout << "[LOG] " << msg << "\n";
    }
};
```

```
void process() {
    auto logger = std::make_unique<Logger>(); // ownership is automatic
    logger->log("Processing started...");

    if (true) {
        std::cout << "An error occurred!\n";
        return; // automatic cleanup, no memory leak
    }
}

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

Improvements:

- Memory is automatically released when logger goes out of scope.
- The code is exception-safe and self-documenting.
- Ownership is clear and explicit — logger is local to process().

- Advanced Example: Non-Owning Raw Pointer (Observer Only)

  In Modern C++, raw pointers are acceptable only when they do not manage ownership, for example, as observers or references to existing objects.

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

class Device {
public:
    void operate() const { std::cout << "Device operating...\n"; }
};
```

```cpp
class Controller {
    Device* device_; // non-owning pointer (observer)
public:
    explicit Controller(Device* dev) : device_(dev) {}
    void control() const {
        if (device_) device_->operate();
    }
};

int main() {
    auto dev = std::make_unique<Device>(); // owns the Device
    Controller ctrl(dev.get());            // observes only
    ctrl.control();
} // Device destroyed automatically when dev goes out of scope
```

Highlights:

- Controller doesn't own the Device; it just references it.
- Ownership and destruction remain centralized in std::unique_ptr.
- Clean and predictable lifetime management.

8.2.4 Best Practices for Clean Memory Management

1. Never use new or delete directly — use std::make_unique or std::make_shared.
2. Avoid naked raw pointers for ownership — use smart pointers instead.
3. Use references or raw pointers only for observation, never for ownership transfer.
4. Prefer stack allocation unless dynamic allocation is required.
5. Document ownership semantics clearly when smart pointers are shared among components.

8.2.5 Summary

Avoiding raw pointers is not merely a stylistic choice—it's a fundamental clean code discipline in C++20 and C++23. By replacing raw ownership with smart pointers and clear ownership models, you ensure that your code is:

- Memory-leak resistant
- Exception-safe
- Readable and maintainable
- Aligned with modern C++ best practices

The era of manual memory management is effectively over in Modern C++. Embrace smart pointers and RAII to write safe, robust, and professional-grade software.

8.3 Resource Management Best Practices

Resource management in Modern C++ extends far beyond memory allocation. It encompasses any system resource that requires acquisition and release — such as file handles, sockets, mutexes, database connections, or GPU buffers. The essence of modern C++ design is that resource lifetime must be deterministic and exception-safe, achieved through RAII (Resource Acquisition Is Initialization), smart pointers, and automatic scope management.
In C++20 and C++23, these principles have become even more powerful due to improvements in move semantics, constexpr constructors, and range-based lifetime control, making clean and safe resource management more natural than ever.

8.3.1 The Core Idea: Ownership and Lifetime

Every resource should have a single, clear owner responsible for releasing it.
C++ provides three common ownership patterns:

- Exclusive ownership using std::unique_ptr
- Shared ownership using std::shared_ptr and std::weak_ptr
- Non-owning access through raw pointers or references

When ownership is explicit, resource release becomes predictable and free from leaks or premature destruction.

8.3.2 The Problem: Manual Resource Management

Manual management using raw pointers or direct API handles is error-prone. Common issues include:

- Resource leaks when forgetting to release handles.
- Double release when the same resource is deleted twice.
- Exception unsafety, where resource cleanup is skipped due to early exit or thrown exceptions.
- Unclear lifetime semantics, making maintenance difficult.

8.3.3 Example: Bad Code — Manual Resource Handling

```
#include <fstream>
#include <string>

void writeToFile(const std::string& path, const std::string& text) {
    std::ofstream* file = new std::ofstream(path); // manual resource allocation
    if (!file->is_open()) {
        delete file; // manual cleanup
        return;
    }

    (*file) << text; // using the file
    // Forgot to delete file -> resource leak
}

int main() {
    writeToFile("data.txt", "Hello World!");
}
```

Problems:

- Uses dynamic allocation for a file stream unnecessarily.
- Potential resource leak if an exception occurs before delete.

- No clear ownership or scope management.
- Function violates RAII principles and lacks exception safety.

8.3.4 Clean Code: RAII and Automatic Scope Control

Modern C++ provides deterministic destruction of local objects at scope exit. This ensures that resources are always cleaned up, regardless of return paths or exceptions.

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

void writeToFile(const std::string& path, const std::string& text) {
    std::ofstream file(path); // automatic resource management
    if (!file.is_open()) {
        std::cerr << "Failed to open file: " << path << "\n";
        return;
    }

    file << text; // safe usage
    // file is automatically closed when leaving scope
}

int main() {
    writeToFile("data.txt", "Hello Clean C++ World!");
}
```

Improvements:

- No manual new or delete.
- Automatic cleanup at scope exit.

- Exception-safe by design.
- Resource management is local, readable, and reliable.

8.3.5 Example: Managing Multiple Resources Safely

A common pitfall arises when managing multiple resources that depend on one another, such as a file and a lock. Modern C++ patterns allow safe composition through RAII and smart objects.

- Bad Example:

```
#include <fstream>
#include <mutex>

std::mutex* fileLock = new std::mutex();

void unsafeWrite(const std::string& data) {
    fileLock->lock();                  // manual lock
    std::ofstream* file = new std::ofstream("log.txt");
    (*file) << data;                   // write
    delete file;                     // cleanup
    fileLock->unlock();                // potential deadlock if exception occurs
}
```

  Issues:

  - Manual locking/unlocking — risk of deadlock.
  - Memory leak risk if exceptions are thrown.
  - Multiple raw pointers obscure ownership semantics.

- Clean Example: Using RAII with Smart Locking and Automatic Cleanup

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

std::mutex fileLock;

void safeWrite(const std::string& data) {
    std::lock_guard<std::mutex> guard(fileLock); // RAII lock
    std::ofstream file("log.txt", std::ios::app);
    if (!file.is_open()) return;

    file << data << '\n'; // file and lock released automatically
}
```

Improvements:

- Lock is automatically released when guard goes out of scope.
- File is automatically closed when file goes out of scope.
- Exception-safe and easy to reason about.
- Ownership and lifetime are explicit, simple, and clean.

8.3.6 C++20/23 Best Practices for Resource Management

1. Apply RAII universally — use automatic cleanup for every resource type (files, sockets, mutexes, etc.).
2. Prefer smart pointers (unique_ptr, shared_ptr) over manual memory handling.
3. Use scope-bound resource management such as std::lock_guard, std::scoped_lock, or custom RAII wrappers.
4. Leverage move semantics to transfer ownership safely without copies.

5. Avoid naked new or delete — instead use factory functions like std::make_unique and std::make_shared.

6. Use constexpr destructors and constructors (C++20/23) when creating lightweight, compile-time RAII wrappers for deterministic cleanup in constant expressions.

7. Prefer composition over inheritance for resource holders, as composition makes ownership relationships explicit.

8.3.7 Example: Custom RAII Wrapper (Modern Style)

When the standard library lacks a resource type, writing a custom RAII wrapper is simple and safe.

```
#include <cstdio>
#include <stdexcept>
#include <string>

class FileHandler {
    std::FILE* file_ = nullptr;
public:
    explicit FileHandler(const std::string& path, const std::string& mode) {
        file_ = std::fopen(path.c_str(), mode.c_str());
        if (!file_) throw std::runtime_error("Failed to open file");
    }

    ~FileHandler() noexcept {
        if (file_) std::fclose(file_);
    }

    FileHandler(const FileHandler&) = delete;
    FileHandler& operator=(const FileHandler&) = delete;
```

```
    FileHandler(FileHandler&& other) noexcept : file_(other.file_) {
        other.file_ = nullptr;
    }

    FileHandler& operator=(FileHandler&& other) noexcept {
        if (this != &other) {
            if (file_) std::fclose(file_);
            file_ = other.file_;
            other.file_ = nullptr;
        }
        return *this;
    }

    void write(const std::string& text) {
        if (file_) std::fputs(text.c_str(), file_);
    }
};

int main() {
    try {
        FileHandler file("output.txt", "w");
        file.write("RAII-managed file writing.\n");
    } catch (const std::exception& e) {
        std::fprintf(stderr, "Error: %s\n", e.what());
    }
}
```

Key points:

- Manages FILE* safely within scope.
- Fully exception-safe and move-enabled (C++20/23 idiomatic).
- Explicit ownership semantics prevent leaks or misuse.

8.3.8 Summary

Modern C++ resource management revolves around automatic lifetime handling, explicit ownership, and exception safety.
C++20 and C++23 reinforce these principles through stronger move semantics, constexpr support, and richer standard utilities.
By following these practices:

- Resource leaks are eliminated.
- Ownership is transparent.
- Code becomes robust, concise, and maintainable.

Clean resource management is not only a coding habit — it is a core philosophy of Modern C++ design.

Chapter 9

Clean Parallelism and Concurrency

9.1 Using std::thread and std::async Safely

Parallelism and concurrency are essential in modern C++ for utilizing multicore architectures efficiently. However, improper thread management often leads to data races, resource leaks, and undefined behavior. Clean code in this area requires disciplined use of synchronization mechanisms, lifetime management, and structured concurrency principles.
C++20 and C++23 introduced stronger guarantees, better synchronization primitives, and clearer patterns for asynchronous execution, allowing developers to write safer and more maintainable concurrent code.

9.1.1 The Problem with Raw Thread Management

A common source of error in concurrent programs is the unsafe creation and management of threads using std::thread directly without proper synchronization or exception safety.

Bad Code Example – Unsafe Thread Usage

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

void worker(int id) {
    std::cout << "Worker " << id << " is running\n";
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 5; ++i)
        threads.emplace_back(worker, i); // Launch threads

    // Forgot to join or detach — undefined behavior!
    // Threads may still be running when main exits.
    return 0;
}
```

Issues:

- The program may terminate while threads are still running, leading to undefined behavior.
- Lack of exception safety: if an exception occurs before joining, the destructor of std::thread will call std::terminate.
- No control over thread synchronization or resource cleanup.

9.1.2 Clean Code Example – Safe and Structured Thread Usage

To write clean, safe concurrent code, threads should always be joined or detached, and exceptions must be handled gracefully. RAII-based management ensures proper cleanup.

Clean Code Example – RAII Thread Guard

```
#include <thread>
#include <iostream>
#include <vector>
#include <utility>

class ThreadGuard {
    std::vector<std::thread>& threads;
public:
    explicit ThreadGuard(std::vector<std::thread>& t) : threads(t) {}
    ~ThreadGuard() {
        for (auto& t : threads)
            if (t.joinable())
                t.join();
    }
    ThreadGuard(const ThreadGuard&) = delete;
    ThreadGuard& operator=(const ThreadGuard&) = delete;
};

void worker(int id) {
    std::cout << "Worker " << id << " is running\n";
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 5; ++i)
        threads.emplace_back(worker, i);

    ThreadGuard guard(threads); // Ensures safe joining
    std::cout << "All threads launched safely.\n";
}
```

Key improvements:

- Automatic joining via RAII ensures thread completion.
- Exception-safe and resource-conscious.
- Clean structure separating logic from synchronization.

9.1.3 Leveraging std::async for Simpler Asynchronous Execution

C++11 introduced std::async, which automatically manages thread creation and synchronization using futures. C++20 improved performance consistency and allowed integration with coroutines.

Bad Code Example – Manual Thread for Computation

```
#include <thread>
#include <iostream>

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

int main() {
    int result = 0;
    std::thread t([&] { result = compute(10); });
    t.join();
    std::cout << "Result: " << result << '\n';
}
```

Issues:

- Shared mutable state (result) risks data races in more complex cases.
- Thread management is manual, error-prone, and lacks flexibility.

Clean Code Example – Using std::async Safely

```
#include <future>
#include <iostream>

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

int main() {
    auto future = std::async(std::launch::async, compute, 10);
    int result = future.get(); // Blocks until ready
    std::cout << "Result: " << result << '\n';
}
```

Advantages:

- Automatic synchronization: no need for explicit joins.
- No shared mutable state; return values handled via futures.
- Exception-safe: exceptions in the async task are propagated to future::get().

9.1.4 Best Practices for Clean Concurrency

1. Prefer std::async over manual threads when you need task-level concurrency, not thread-level control.
2. Always manage thread lifetimes explicitly or through RAII wrappers.
3. Avoid shared mutable data; use std::mutex, std::lock_guard, or immutable objects when necessary.

4. Use structured concurrency principles (proposed for C++26) to manage task hierarchies predictably.

5. Test and profile concurrent code under real workloads—logical correctness does not imply performance scalability.

9.1.5 Summary

Clean concurrency in modern C++ is about predictability, safety, and minimalism. C++20 and C++23 provide the tools needed to express parallel logic clearly while avoiding the pitfalls of manual thread management.
std::thread remains powerful for explicit control, while std::async offers automatic synchronization and exception safety for simpler asynchronous workflows. The hallmark of professional C++ concurrency is not how many threads are spawned, but how safely and clearly they are managed.

9.2 Atomic Operations and Mutex – Clean Practices

Concurrency in C++ introduces the challenge of managing shared resources safely. Without proper synchronization, race conditions and inconsistent states can occur. Modern C++ provides low-level tools like atomic operations (std::atomic) and higher-level synchronization primitives such as mutexes (std::mutex, std::shared_mutex, and std::scoped_lock) to ensure thread safety and predictable behavior.
Writing clean concurrent code is not about using these primitives everywhere, but about using them deliberately and minimally, favoring immutability and confinement of shared state whenever possible.

9.2.1 Common Mistakes: Unsafe Shared Data Access

A typical error occurs when multiple threads modify shared data without synchronization, causing undefined behavior even in seemingly simple operations.

Bad Code Example – Race Condition on Shared Variable

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

int counter = 0;

void increment() {
    for (int i = 0; i < 1000; ++i)
        ++counter; // Not atomic! Race condition possible
}

int main() {
    std::vector<std::thread> threads;
```

```
    for (int i = 0; i < 4; ++i)
        threads.emplace_back(increment);

    for (auto& t : threads)
        t.join();

    std::cout << "Counter = " << counter << '\n'; // Unpredictable result
}
```

Issues:

- Multiple threads modify counter concurrently without synchronization.
- Results vary unpredictably between runs.
- Hard to debug and may pass casual testing despite incorrect behavior.

9.2.2 Clean Code Example – Using std::atomic for Simple Synchronization

When you need to synchronize simple arithmetic or boolean flags, use atomic operations. std::atomic ensures lock-free and well-defined operations on shared data, when supported by the hardware.

Clean Code Example – Safe Atomic Counter

```
#include <atomic>
#include <thread>
#include <iostream>
#include <vector>

std::atomic<int> counter = 0;

void increment() {
```

```
    for (int i = 0; i < 1000; ++i)
        counter.fetch_add(1, std::memory_order_relaxed);
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i)
        threads.emplace_back(increment);

    for (auto& t : threads)
        t.join();

    std::cout << "Counter = " << counter.load() << '\n'; // Deterministic result
}
```

Key improvements:

- std::atomic provides well-defined synchronization.
- No need for explicit locks for simple operations.
- Memory ordering (std::memory_order_relaxed) improves performance when strict ordering is unnecessary.

Note:
While atomics are efficient, they are not a replacement for mutexes when managing complex shared data. Using atomics for compound operations on multiple variables often leads to subtle bugs and unreadable code.

9.2.3 Clean Code Example – Using std::mutex and RAII for Shared Data

When multiple operations must be performed as a single logical unit, mutexes provide a safer abstraction. The key is to manage locks cleanly, ensuring they are always released—even in the presence of exceptions.

Bad Code Example – Manual Lock Management

```
#include <mutex>
#include <thread>
#include <iostream>
#include <vector>

int shared_value = 0;
std::mutex mtx;

void update_value() {
    mtx.lock(); // Manual locking
    shared_value += 10;
    // Forgot to unlock if an exception occurs!
    mtx.unlock();
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 5; ++i)
        threads.emplace_back(update_value);

    for (auto& t : threads)
        t.join();

    std::cout << "Shared value = " << shared_value << '\n';
}
```

Issues:

- Manual lock/unlock leads to potential deadlocks or leaks if exceptions occur.
- Not exception-safe and violates RAII principles.

Clean Code Example – Using RAII with std::lock_guard

```
#include <mutex>
#include <thread>
#include <iostream>
#include <vector>

int shared_value = 0;
std::mutex mtx;

void update_value() {
    std::lock_guard<std::mutex> lock(mtx); // RAII lock
    shared_value += 10;
    // Lock released automatically when 'lock' goes out of scope
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 5; ++i)
        threads.emplace_back(update_value);

    for (auto& t : threads)
        t.join();

    std::cout << "Shared value = " << shared_value << '\n';
}
```

Key improvements:

- std::lock_guard guarantees automatic unlock even on exceptions.
- Code is concise, readable, and exception-safe.
- Resource ownership follows RAII principles, ensuring safety and clarity.

9.2.4 Clean Code Example – std::scoped_lock and Multiple Mutexes

C++17 introduced std::scoped_lock to simplify locking multiple mutexes safely without deadlocks. This feature remains highly relevant in C++20 and C++23 for clean and deterministic locking behavior.

```
#include <mutex>
#include <thread>
#include <iostream>
#include <vector>

int shared_a = 0;
int shared_b = 0;
std::mutex mtx_a, mtx_b;

void transfer(int value) {
    std::scoped_lock lock(mtx_a, mtx_b); // Deadlock-free locking of both
    shared_a -= value;
    shared_b += value;
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 5; ++i)
        threads.emplace_back(transfer, 5);

    for (auto& t : threads)
        t.join();

    std::cout << "A = " << shared_a << ", B = " << shared_b << '\n';
}
```

Clean advantages:

- Prevents deadlock by locking all mutexes atomically.
- Provides strong exception safety.
- Readable and consistent with modern RAII conventions.

9.2.5 Best Practices for Clean Synchronization in Modern C++

1. Prefer immutability and confinement. The safest shared data is one that is never shared.
2. Use std::atomic only for trivial synchronization (counters, flags, single variables).
3. Use std::mutex for complex or multi-variable updates. Combine with RAII (std::lock_guard, std::scoped_lock).
4. Avoid manual lock/unlock. Let the type system manage scope and lifetime.
5. Use the narrowest scope possible for locks to minimize contention.
6. Never mix atomic and mutex-protected access to the same data — it leads to undefined behavior.

9.2.6 Summary

C++20 and C++23 enable safe and expressive concurrency through a mature synchronization model.
std::atomic ensures fine-grained, lock-free synchronization for simple cases, while std::mutex and its RAII wrappers provide structured control for complex scenarios.
Clean C++ concurrency is defined not by how aggressively threads are used, but by how predictably data integrity is maintained. Professional developers ensure that shared

data is protected, lifetime-managed, and scoped tightly, allowing parallelism without compromising correctness or readability.

9.3 Parallel STL Algorithms in C++20/23

Parallelism in Modern C++ has evolved significantly with the introduction of Parallel Algorithms in C++17, refined and extended in C++20 and C++23. These algorithms leverage execution policies to parallelize computations safely and efficiently, allowing developers to exploit multicore architectures without resorting to manual thread management.
C++20 and C++23 continue to refine the parallel execution model, ensuring better consistency, performance portability, and integration with ranges. Clean C++ code leverages these features to simplify concurrency while maintaining determinism and clarity.

9.3.1 The Problem: Manual Thread Management and Complexity

Before parallel algorithms, programmers often wrote explicit multithreaded loops to utilize CPU cores. This approach introduces verbosity, synchronization risks, and maintenance challenges.

Bad Code Example – Manual Threading for Parallel Work

```
#include <thread>
#include <vector>
#include <numeric>
#include <iostream>

int main() {
    std::vector<int> data(1000000);
    std::iota(data.begin(), data.end(), 1);

    auto sum_part = [&](size_t start, size_t end, long long& result) {
        result = std::accumulate(data.begin() + start, data.begin() + end, 0LL);
```

```
    };

    long long result1 = 0, result2 = 0, result3 = 0, result4 = 0;
    size_t size = data.size() / 4;

    std::thread t1(sum_part, 0, size, std::ref(result1));
    std::thread t2(sum_part, size, 2 * size, std::ref(result2));
    std::thread t3(sum_part, 2 * size, 3 * size, std::ref(result3));
    std::thread t4(sum_part, 3 * size, data.size(), std::ref(result4));

    t1.join(); t2.join(); t3.join(); t4.join();

    long long total = result1 + result2 + result3 + result4;
    std::cout << "Total sum = " << total << '\n';
}
```

Issues:

- Manual thread management and partitioning are error-prone.
- No scalability — requires adjusting thread count manually.
- Code is verbose and hard to maintain.
- Lacks exception safety and synchronization clarity.

9.3.2 Clean Code Example – Using Parallel STL Algorithms

C++20 allows replacing complex thread code with a single, expressive call using execution policies defined in <execution>.

Clean Code Example – Using std::reduce with std::execution::par

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

int main() {
    std::vector<int> data(1'000'000);
    std::iota(data.begin(), data.end(), 1);

    long long total = std::reduce(std::execution::par, data.begin(), data.end(), 0LL);

    std::cout << "Total sum = " << total << '\n';
}
```

Advantages:

- No manual threading — parallelism is handled by the implementation.
- Safe and deterministic reduction (associative operations required).
- Easily scalable across multiple cores.
- Cleaner and shorter code.

Execution Policies:

- std::execution::seq: Sequential (default).
- std::execution::par: Parallel execution (may use multiple threads).
- std::execution::par_unseq: Parallel and vectorized execution.
- std::execution::unseq: Vectorized execution only.

9.3.3 Writing Clean Parallel Code Using C++20/23 Ranges

C++20 introduced Ranges, and C++23 improves their interaction with execution policies, allowing even more expressive and clean data processing pipelines.

Clean Code Example – Combining Ranges and Parallel Algorithms

```
#include <vector>
#include <numeric>
#include <execution>
#include <ranges>
#include <iostream>

int main() {
    std::vector<int> data(1'000'000);
    std::iota(data.begin(), data.end(), 1);

    // Compute the sum of squares in parallel using ranges
    auto total = std::transform_reduce(
        std::execution::par,
        data.begin(), data.end(),
        0LL,
        std::plus<>(),
        [](int x) { return x * x; }
    );

    std::cout << "Sum of squares = " << total << '\n';
}
```

Key advantages:

- Declarative and readable: Expresses intent instead of mechanics.
- Efficient: Implementation decides optimal thread usage.

- Composable: Integrates with Ranges transformations and views.
- Exception-safe: Managed by the standard library with clear guarantees.

9.3.4 Common Mistakes When Using Parallel STL Algorithms

1. Non-associative operations:
   Parallel reductions assume associativity. Using non-associative operations like floating-point subtraction can lead to inconsistent results.
2. Modifying shared data:
   Algorithms like std::for_each(std::execution::par, ...) should not modify shared or global variables without synchronization.
3. Mixing sequential and parallel code unsafely:
   Keep data ownership clear—parallel algorithms must work on disjoint or read-only ranges.
4. Ignoring exception safety:
   When exceptions occur in parallel execution, they propagate as std::exception_list. Handle these explicitly.

9.3.5 Clean Code Example – Exception-Safe Parallel Transformation

```
#include <vector>
#include <execution>
#include <iostream>
#include <exception>

int main() {
    std::vector<int> data = {1, 2, 3, -4, 5};
    std::vector<int> results(data.size());
```

```
    try {
        std::transform(std::execution::par, data.begin(), data.end(), results.begin(),
                       [](int x) {
                           if (x < 0) throw std::runtime_error("Negative value!");
                           return x * 2;
                       });
    } catch (const std::exception_list& el) {
        for (const auto& e : el) {
            try { if (e) std::rethrow_exception(e); }
            catch (const std::exception& ex) {
                std::cerr << "Exception: " << ex.what() << '\n';
            }
        }
    }

    for (int val : results)
        std::cout << val << " ";
}
```

Explanation:

- C++ standard ensures all exceptions from worker threads are collected into a std::exception_list.
- Each can be safely handled, preserving program stability.
- Cleanly structured with clear error reporting and parallel safety.

9.3.6 Clean Parallelism Best Practices in Modern C++

1. Use parallel algorithms before manual threading. They're cleaner, safer, and portable.

2. Choose the right execution policy. Use par for multi-threading and par_unseq for vectorized computation when safe.
3. Keep operations associative and independent. Avoid shared state.
4. Prefer pure functions for transformation and reduction steps.
5. Test performance carefully. Parallel execution isn't always faster for small datasets.
6. Use ranges for cleaner pipelines when available (C++20/23).

9.3.7 Summary

Parallel STL algorithms represent the culmination of decades of concurrency refinement in C++, merging safety, clarity, and performance. With execution policies, RAII-based thread management, and functional-style algorithms, developers can write scalable parallel code without the pitfalls of manual synchronization.
In C++20 and C++23, clean parallelism means writing expressive, data-centric code — allowing the compiler and runtime to manage execution details efficiently.
By combining parallel algorithms with ranges and pure functions, C++ programmers achieve both clarity and high performance — the essence of modern clean C++ design.

Chapter 10

Modern C++ Features and Best Practices

10.1 Concepts, Modules, Coroutines

Modern C++ (C++20/23) introduces powerful language features that improve code clarity, safety, modularity, and performance. Among these, Concepts, Modules, and Coroutines represent major advancements in writing clean, maintainable, and expressive C++ code. Understanding these features is essential for professional C++ development.

10.1.1 Concepts – Type-Safe and Expressive Templates

Before C++20, templates often produced obscure error messages when type requirements were not met. Concepts provide compile-time constraints, enabling self-documenting, readable, and safe generic code.

Bad Code Example – Classic Template Without Concepts

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

```
template<typename T>
T sum(const std::vector<T>& vec) {
    T result = 0;
    for (const auto& v : vec)
        result += v; // Works only if T supports operator+
    return result;
}

int main() {
    std::vector<int> nums = {1, 2, 3, 4};
    std::cout << sum(nums) << '\n';

    std::vector<std::string> strings = {"a", "b", "c"};
    std::cout << sum(strings) << '\n'; // Compilation error is cryptic
}
```

Issues:

- No clear requirement for template parameter T.
- Compilation errors are verbose and difficult to debug.

Clean Code Example – Using Concepts

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

template<std::totally_ordered T>
T sum(const std::vector<T>& vec) {
    T result = 0;
    for (const auto& v : vec)
```

```
        result += v;
    return result;
}

int main() {
    std::vector<int> nums = {1, 2, 3, 4};
    std::cout << sum(nums) << '\n';
}
```

Advantages:

- Constraints communicate requirements clearly (std::totally_ordered).
- Cleaner error messages for unsupported types.
- Improves readability and maintainability.

10.1.2 Modules – Cleaner Project Organization

Modules, introduced in C++20, replace the traditional header/include model, reducing compilation times and minimizing macro-related problems.

Bad Code Example – Traditional Header Includes

```
// math_utils.h
int add(int a, int b);

// main.cpp
#include "math_utils.h"
#include <iostream>
int main() {
    std::cout << add(2, 3) << "\n";
}
```

Issues:

- Multiple inclusions can slow compilation.
- Macros and preprocessor errors can pollute global namespace.
- Dependencies are hard to track in large projects.

Clean Code Example – Using Modules

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

// main.cpp
import math_utils;
#include <iostream>
int main() {
    std::cout << add(2, 3) << "\n";
}
```

Advantages:

- No need for include guards.
- Explicitly exported interfaces improve readability.
- Faster compilation and modular project structure.

10.1.3 Coroutines – Efficient Asynchronous Programming

C++20 introduced coroutines, allowing functions to suspend and resume execution without blocking threads. They provide a cleaner alternative to callbacks and manual state machines.

Bad Code Example – Manual Thread for Delayed Operation

```
#include <thread>
#include <iostream>

void delayed_print() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Hello after delay\n";
}

int main() {
    std::thread t(delayed_print);
    t.join();
}
```

Issues:

- Manual thread management for simple asynchronous tasks.
- Not scalable for many concurrent operations.

Clean Code Example – Using Coroutines

```
#include <coroutine>
#include <iostream>
#include <thread>
#include <chrono>
```

```
struct Task {
    struct promise_type {
        Task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() {}
    };
};

Task delayed_print() {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(1s);
    std::cout << "Hello after delay\n";
    co_return;
}

int main() {
    auto t = delayed_print(); // Non-blocking, coroutine executed
}
```

Advantages:

- Simplifies asynchronous code by removing manual thread management.
- Clean, readable syntax for sequential logic.
- Enables scalable async programming with minimal boilerplate.

10.1.4 Best Practices for Clean Modern C++

1. Use Concepts for generic code. Clearly express constraints to avoid ambiguous template errors.

2. Prefer Modules for large projects. Modular design improves compilation speed and maintainability.

3. Adopt Coroutines for asynchronous workflows. Replace callback-heavy or manual-thread code with sequential, readable coroutines.

4. Combine modern features. For instance, a module can export a coroutine-based API constrained by concepts for maximum clarity and safety.

5. Keep code minimal and expressive. Avoid mixing old preprocessor-based patterns with modern features to maintain clean code.

10.1.5 Summary

Concepts, Modules, and Coroutines are pillars of clean, modern C++.

- Concepts make generic code self-documenting and type-safe.
- Modules replace headers for modular and fast compilation.
- Coroutines provide clear, asynchronous code without threading complexity.

Using these features thoughtfully ensures clarity, maintainability, and professional-quality code while fully leveraging C++20 and C++23 capabilities.

10.2 std::format, std::span, std::bit_cast

C++20 and C++23 introduce several modern utilities that enhance code readability, safety, and expressiveness. Among these, std::format, std::span, and std::bit_cast are essential tools for writing clean, professional C++ code.

10.2.1 std::format – Safe and Readable String Formatting

Before C++20, printf and std::ostringstream were common, but each had issues: type safety, verbosity, and readability.

Bad Code Example – Classic Formatting

```
#include <iostream>
#include <cstdio>

int main() {
    int x = 42;
    double y = 3.14159;
    char buffer[50];
    std::sprintf(buffer, "x = %d, y = %.2f", x, y); // Unsafe, may overflow
    std::cout << buffer << '\n';
}
```

Issues:

- Unsafe buffer usage (sprintf).
- Type mismatches can cause undefined behavior.
- Verbose and less readable.

Clean Code Example – Using std::format

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

int main() {
    int x = 42;
    double y = 3.14159;
    std::cout << std::format("x = {}, y = {:.2f}\n", x, y);
}
```

Advantages:

- Type-safe formatting.
- Clear, concise syntax.
- Flexible formatting options (width, precision, alignment).

10.2.2 std::span – Safe Views over Arrays and Containers

std::span (C++20) allows non-owning views over sequences, replacing raw pointers and manual size tracking.

Bad Code Example – Raw Pointer Array Access

```
#include <iostream>

void print_array(int* arr, size_t size) {
    for (size_t i = 0; i < size; ++i)
        std::cout << arr[i] << " ";
    std::cout << '\n';
}

int main() {
```

```
    int nums[] = {1, 2, 3, 4};
    print_array(nums, 4);
}
```

Issues:

- No size safety guarantees.
- Hard to integrate with modern containers.
- Less expressive about intent.

Clean Code Example – Using std::span

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

void print_array(std::span<const int> arr) {
    for (int value : arr)
        std::cout << value << " ";
    std::cout << '\n';
}

int main() {
    std::vector<int> nums = {1, 2, 3, 4};
    print_array(nums); // Works for arrays, vectors, or subranges
}
```

Advantages:

- Safer, self-documenting view of data.
- Works with arrays, vectors, and other contiguous containers.
- Does not copy data — efficient and clean.

10.2.3 std::bit_cast – Safe Bitwise Type Reinterpretation

C++20 introduces std::bit_cast to safely reinterpret object representations without undefined behavior, replacing reinterpret_cast in some cases.

Bad Code Example – Unsafe Type Punning

```
#include <iostream>

int main() {
    float f = 3.14f;
    int i = *reinterpret_cast<int*>(&f); // Undefined behavior
    std::cout << i << '\n';
}
```

Issues:

- Undefined behavior according to strict aliasing rules.
- Non-portable and error-prone.

Clean Code Example – Using std::bit_cast

```
#include <iostream>
#include <bit>
#include <cstdint>

int main() {
    float f = 3.14f;
    std::uint32_t i = std::bit_cast<std::uint32_t>(f);
    std::cout << i << '\n';
}
```

Advantages:

- Well-defined and safe type reinterpretation.
- Expresses intent clearly.
- Portable across platforms.

10.2.4 Best Practices for Clean Modern C++

1. Prefer std::format over printf or ostringstream. It is safer, cleaner, and easier to read.
2. Use std::span for function parameters instead of raw pointers. This avoids manual size errors and improves expressiveness.
3. Replace unsafe type-punning with std::bit_cast. Only use it for trivially copyable types to preserve strict aliasing rules.
4. Combine these features for clarity. For instance, std::format can take values from a std::span to print sequences safely.
5. Write minimal, expressive code. Avoid verbose workarounds or legacy patterns that modern C++ now handles safely.

10.2.5 Summary

C++20 and C++23 features like std::format, std::span, and std::bit_cast enable cleaner, safer, and more expressive code. These utilities remove common sources of bugs in string formatting, memory access, and low-level type manipulation, aligning with the core principles of modern clean C++ design.
By embracing these features, developers can write maintainable, readable, and high-performance code without sacrificing clarity or safety.

10.3 Writing Clean, Maintainable Coroutines

Coroutines in C++20 provide a powerful mechanism for asynchronous programming, generators, and cooperative multitasking. They allow functions to suspend and resume execution, which can greatly simplify complex logic. Writing clean and maintainable coroutines is crucial to leverage their full potential without introducing subtle bugs or complexity.

10.3.1 The Problem: Poorly Structured Coroutines

Without careful design, coroutines can become hard to read, maintain, and debug, especially when manually managing suspensions and shared state.

Bad Code Example – Unstructured Coroutine

```
#include <coroutine>
#include <iostream>
#include <thread>
#include <vector>

struct Task {
    struct promise_type {
        Task get_return_object() { return {}; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() {}
    };
};

Task print_numbers() {
    for (int i = 1; i <= 5; ++i) {
```

```
        std::cout << i << "\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(500)); // blocking
        co_await std::suspend_always{};
    }
}

int main() {
    auto t = print_numbers();
}
```

Issues:

- Blocking sleep_for inside coroutine defeats asynchronous benefits.
- Suspension logic is manually interleaved with business logic.
- Hard to reuse or compose with other coroutines.

10.3.2 Clean Code Example – Structured, Maintainable Coroutine

Modern C++ design encourages separating suspension mechanics from business logic, and leveraging coroutine-friendly utilities such as co_await with awaitables or generator-style coroutines.

Clean Code Example – Non-blocking Generator Coroutine

```
#include <coroutine>
#include <iostream>
#include <optional>

template<typename T>
struct Generator {
    struct promise_type {
```

```
        T current_value;
        std::suspend_always yield_value(T value) {
            current_value = value;
            return {};
        }
        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 {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    std::optional<T> next() {
        if (!handle.done()) {
            handle.resume();
            return handle.promise().current_value;
        }
        return std::nullopt;
    }
};

Generator<int> generate_numbers(int n) {
    for (int i = 1; i <= n; ++i)
        co_yield i;
}

int main() {
```

```
    auto gen = generate_numbers(5);
    while (auto value = gen.next()) {
        std::cout << *value << "\n";
    }
}
```

Advantages:

- Clear separation between suspension and business logic (co_yield expresses intent).
- Non-blocking; can be easily integrated with async frameworks.
- Reusable generator abstraction.
- Exception safety is explicit and contained.

10.3.3 Best Practices for Clean Coroutines

1. Separate logic from suspension
    - Avoid embedding sleep, blocking calls, or I/O directly in coroutine body.
    - Use co_await on proper awaitables to keep coroutines composable.
2. Use generators for iterative sequences
    - co_yield clearly expresses data production without side effects.
3. Avoid raw coroutine handles in application logic
    - Encapsulate in types like Generator, Task, or Future to maintain RAII safety.
4. Handle exceptions explicitly

- Ensure unhandled_exception is defined. Use try/catch in top-level coroutine consumers.

5. Minimize side effects

   - Keep coroutine logic pure where possible; side effects should occur in consumer code.

6. Leverage modern C++20/23 features

   - Combine coroutines with std::span, std::format, and ranges for clearer and efficient pipelines.

10.3.4 Example: Clean Asynchronous Pipeline with Coroutines

```
#include <coroutine>
#include <iostream>
#include <vector>
#include <optional>

template<typename T>
struct Generator {
    struct promise_type {
        T current_value;
        std::suspend_always yield_value(T value) {
            current_value = value;
            return {};
        }
        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 {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    std::optional<T> next() {
        if (!handle.done()) {
            handle.resume();
            return handle.promise().current_value;
        }
        return std::nullopt;
    }
};

Generator<int> filter_even(const std::vector<int>& data) {
    for (int value : data) {
        if (value % 2 == 0) co_yield value;
    }
}

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
    auto gen = filter_even(numbers);

    while (auto value = gen.next()) {
        std::cout << *value << " ";
    }
}
```

Highlights of Clean Design:

- Declarative, expressive coroutine (co_yield) for filtering.
- No manual state or blocking operations.
- Easily extendable to other filters, pipelines, or async tasks.

10.3.5 Summary

Writing clean coroutines in C++20/23 means creating modular, readable, and reusable async code. Key principles:

- Separate suspension mechanics from business logic.
- Use generators for sequences and awaitables for async operations.
- Ensure exception safety and RAII-based handle management.
- Combine coroutines with other modern C++20/23 features for maximum clarity and maintainability.

Coroutines, when written cleanly, enable scalable asynchronous workflows while maintaining the clarity and safety expected of professional C++ code.

10.4 Practical Examples of Modern Features Used Cleanly

Chapter 10: Modern C++ Features and Best Practices
Booklet: *C++ Clean Code: The Definitive Practical Guide (C++20 & C++23)*
C++20 and C++23 introduce powerful features that, when used correctly, greatly enhance code clarity, safety, and maintainability. Practical examples demonstrate how modern features can replace legacy patterns with clean, professional, and expressive code.

10.4.1 Combining std::format, std::span, and Ranges

Bad Code Example – Legacy Style

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

int main() {
    std::vector<int> numbers = {1, 2, 3, 4};
    for (size_t i = 0; i < numbers.size(); ++i) {
        char buffer[20];
        std::sprintf(buffer, "Value: %d\n", numbers[i]);
        std::cout << buffer;
    }
}
```

Issues:

- Unsafe sprintf usage.
- Manual indexing, verbose code.

- Harder to read and maintain.

Clean Code Example – Modern C++20 Style

```
#include <iostream>
#include <vector>
#include <span>
#include <format>
#include <ranges>

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

    // Using std::span for safe view
    std::span<const int> nums_view(numbers);

    // Using ranges and structured for loop
    for (int value : nums_view | std::views::all) {
        std::cout << std::format("Value: {}\n", value);
    }
}
```

Advantages:

- Safe, concise, and readable.
- Avoids raw arrays or manual indexing.
- Type-safe formatting with std::format.
- Fully compatible with C++20 ranges and views.

10.4.2 Using std::bit_cast for Safe Type Reinterpretation

Bad Code Example – Unsafe Type Punning

```
#include <iostream>

int main() {
    float f = 3.14159f;
    int i = *reinterpret_cast<int*>(&f); // Undefined behavior
    std::cout << i << '\n';
}
```

Clean Code Example – Using std::bit_cast

```
#include <iostream>
#include <bit>
#include <cstdint>

int main() {
    float f = 3.14159f;
    std::uint32_t i = std::bit_cast<std::uint32_t>(f);
    std::cout << i << '\n';
}
```

Advantages:

- Well-defined behavior across platforms.
- Explicit and safe for trivially copyable types.

10.4.3 Coroutines for Lazy Generation

Bad Code Example – Manual Iterator with Blocking

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

void print_even(const std::vector<int>& numbers) {
    for (int n : numbers) {
        if (n % 2 == 0) {
            std::this_thread::sleep_for(std::chrono::milliseconds(500)); // blocking
            std::cout << n << " ";
        }
    }
}

int main() {
    print_even({1,2,3,4,5,6});
}
```

Clean Code Example – Generator Coroutine

```
#include <coroutine>
#include <iostream>
#include <vector>
#include <optional>

template<typename T>
struct Generator {
    struct promise_type {
        T value;
        std::suspend_always yield_value(T v) {
            value = v;
            return {};
        }
        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 {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    std::coroutine_handle<promise_type> handle;
    Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    std::optional<T> next() {
        if (!handle.done()) {
            handle.resume();
            return handle.promise().value;
        }
        return std::nullopt;
    }
};

Generator<int> even_numbers(const std::vector<int>& numbers) {
    for (int n : numbers) {
        if (n % 2 == 0) co_yield n;
    }
}

int main() {
    auto gen = even_numbers({1,2,3,4,5,6});
    while (auto v = gen.next()) {
        std::cout << *v << " ";
    }
}
```

Advantages:

- Clean, non-blocking asynchronous iteration.
- Easy to extend for more filters or transformations.
- Encapsulates state automatically.

10.4.4 Combining Concepts, Templates, and Coroutines

Clean Modern Pipeline

```
#include <coroutine>
#include <concepts>
#include <vector>
#include <iostream>

template<std::integral T>
struct Generator {
    struct promise_type {
        T value;
        std::suspend_always yield_value(T v) {
            value = v;
            return {};
        }
        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 {}; }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };
```

```
    std::coroutine_handle<promise_type> handle;
    Generator(std::coroutine_handle<promise_type> h) : handle(h) {}
    ~Generator() { if (handle) handle.destroy(); }

    std::optional<T> next() {
        if (!handle.done()) {
            handle.resume();
            return handle.promise().value;
        }
        return std::nullopt;
    }
};

Generator<int> filter_even(const std::vector<int>& numbers) {
    for (int n : numbers)
        if (n % 2 == 0) co_yield n;
}

int main() {
    std::vector<int> data = {1,2,3,4,5,6};
    auto gen = filter_even(data);

    while (auto v = gen.next())
        std::cout << v.value() << " ";
}
```

Highlights:

- Concepts ensure type safety in templates.
- Coroutines simplify iteration and state management.
- Clean and expressive modern C++20/23 code.

10.4.5 Key Takeaways

1. Prefer std::format over legacy formatting for clarity and safety.
2. Use std::span for safe, non-owning views on sequences.
3. Apply std::bit_cast instead of unsafe type punning.
4. Write generator-style coroutines to separate state and logic.
5. Combine modern features holistically for maintainable, reusable pipelines.
6. Keep code expressive, minimal, and type-safe, aligning with ISO C++ Core Guidelines.

These examples demonstrate how C++20/23 modern features can transform legacy patterns into clean, professional code suitable for real-world production systems.

Chapter 11

Testing and Verification

11.1 Unit Testing (Catch2, GoogleTest)

Unit testing is a critical practice in modern C++ for verifying correctness, preventing regressions, and ensuring maintainability. Frameworks like Catch2 and GoogleTest (gtest) provide professional tooling for writing clear, repeatable, and expressive tests in C++20/23. Writing clean tests aligns closely with the principles of Clean Code, making tests readable, concise, and maintainable.

11.1.1 The Problem: Poorly Written Tests

Poorly structured tests are often hard to read, brittle, and tightly coupled to implementation details.

Bad Code Example – Ad Hoc Testing

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

```
int sum(const std::vector<int>& v) {
    int result = 0;
    for (int n : v) result += n;
    return result;
}

int main() {
    std::vector<int> data{1, 2, 3};
    int result = sum(data);
    if (result == 6) std::cout << "Pass\n";
    else std::cout << "Fail\n";
}
```

Issues:

- Manual comparison and printing.
- No structured assertion mechanism.
- Difficult to extend with multiple test cases.
- No integration with CI/CD pipelines.

11.1.2 Clean Code Example – Using Catch2

Catch2 provides a lightweight, expressive framework for writing modern unit tests.

```
#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>
#include <vector>

// Function under test
int sum(const std::vector<int>& v) {
```

```
    int result = 0;
    for (int n : v) result += n;
    return result;
}

TEST_CASE("Sum of integers in a vector", "[sum]") {
    SECTION("Positive numbers") {
        std::vector<int> data{1, 2, 3};
        REQUIRE(sum(data) == 6);
    }

    SECTION("Empty vector") {
        std::vector<int> data{};
        REQUIRE(sum(data) == 0);
    }

    SECTION("Negative numbers") {
        std::vector<int> data{-1, -2, -3};
        REQUIRE(sum(data) == -6);
    }
}
```

Advantages:

- Readable test structure using TEST_CASE and SECTION.
- Automatic reporting of failures.
- Easily scalable with new test scenarios.
- Integrates seamlessly with modern C++20/23 projects.

11.1.3 Clean Code Example – Using GoogleTest

GoogleTest offers a rich, industry-standard framework for unit testing with expressive macros.

```
#include <gtest/gtest.h>
#include <vector>

int sum(const std::vector<int>& v) {
    int result = 0;
    for (int n : v) result += n;
    return result;
}

TEST(SumTest, PositiveNumbers) {
    std::vector<int> data{1,2,3};
    EXPECT_EQ(sum(data), 6);
}

TEST(SumTest, EmptyVector) {
    std::vector<int> data{};
    EXPECT_EQ(sum(data), 0);
}

TEST(SumTest, NegativeNumbers) {
    std::vector<int> data{-1,-2,-3};
    EXPECT_EQ(sum(data), -6);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Advantages:

- Structured and expressive assertions (EXPECT_EQ, ASSERT_TRUE).
- Test fixtures for reusable setup/teardown.
- Full integration with CI/CD pipelines.
- Suitable for large-scale professional projects.

11.1.4 Best Practices for Clean Unit Tests

1. Test one concept per test case
    - Keep TEST_CASE or TEST focused and minimal.
2. Use descriptive names
    - Express intent clearly: SumTest_PositiveNumbers is better than Test1.
3. Avoid implementation dependencies
    - Tests should verify behavior, not internal implementation.
4. Automate execution
    - Integrate with CI/CD pipelines to catch regressions early.
5. Leverage modern C++20/23 features
    - Use constexpr, ranges, std::span, and structured bindings inside tests for clearer, safer assertions.

11.1.5 Example of Clean Test Using Modern C++20/23

```
#include <catch2/catch.hpp>
#include <vector>
#include <ranges>

int sum_even(const std::vector<int>& v) {
    int result = 0;
    for (int n : v | std::views::filter([](int x){ return x % 2 == 0; }))
        result += n;
    return result;
}

TEST_CASE("Sum only even numbers", "[sum_even]") {
    std::vector<int> data{1,2,3,4,5,6};
    REQUIRE(sum_even(data) == 12);
}
```

Highlights:

- Clean separation of logic (sum_even) and test.
- Use of ranges for readable and expressive filtering.
- Concise and maintainable test structure.

11.1.6 Summary

Clean unit testing in C++20/23 using frameworks like Catch2 and GoogleTest ensures:

- Tests are readable, concise, and maintainable.
- Behavior is verified clearly without fragile manual checks.

- Modern C++ features can be used inside tests for expressiveness and safety.
- Testing practices integrate with professional development pipelines, supporting CI/CD and robust software quality assurance.

Unit testing is essential for professional C++ projects, aligning with Clean Code principles and enabling safe, maintainable, and production-ready software.

11.2 TDD and Writing Testable Code

Test-Driven Development (TDD) is a core practice in modern professional C++ that ensures code correctness, maintainability, and clarity. Writing testable code goes hand-in-hand with Clean Code principles: code should be modular, decoupled, and expressive, enabling automated testing with minimal effort.

11.2.1 The Problem: Non-Testable Code

Poorly designed code often mixes concerns, has hidden dependencies, and is tightly coupled, making it difficult to test or extend.

Bad Code Example – Non-Testable Design

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

class DataProcessor {
public:
    void process() {
        std::vector<int> data = {1,2,3,4,5};
        int sum = 0;
        for (int n : data) sum += n;
        std::cout << "Sum: " << sum << '\n';
    }
};

int main() {
    DataProcessor dp;
    dp.process(); // Hard to test, prints directly
}
```

Issues:

- Combines data retrieval, computation, and output.
- Hard-coded values prevent reusability.
- No way to assert correctness in automated tests.

11.2.2 Clean Code Example – Testable Design

Refactoring for TDD and testability:

```
#include <vector>
#include <numeric> // For std::accumulate

class DataProcessor {
public:
    // Pure function – easy to test
    int sum(const std::vector<int>& data) const {
        return std::accumulate(data.begin(), data.end(), 0);
    }
};
```

Writing Unit Tests (Catch2 Example)

```
#define CATCH_CONFIG_MAIN
#include <catch2/catch.hpp>

TEST_CASE("Sum computation is correct", "[sum]") {
    DataProcessor dp;
    REQUIRE(dp.sum({1,2,3,4,5}) == 15);
    REQUIRE(dp.sum({}) == 0);
    REQUIRE(dp.sum({-1,-2,3}) == 0);
}
```

Advantages:

- Logic separated from I/O and dependencies.
- Easily tested with multiple scenarios.
- Supports TDD cycles: Red → Green → Refactor.

11.2.3 Principles for Writing Testable Code

1. Separate concerns
    - Keep business logic distinct from I/O or UI.
    - Each function should do one thing.
2. Inject dependencies
    - Pass resources or collaborators via parameters (Dependency Injection).
    - Avoid global state.
3. Use pure functions where possible
    - Functions with no side effects are inherently testable.
4. Leverage modern C++20/23 features
    - std::span for safe sequence handling.
    - constexpr for compile-time evaluation.
    - std::ranges for clean and expressive iteration.

11.2.4 Advanced Example: Testable Modern Pipeline

```
#include <vector>
#include <ranges>
#include <numeric>

class DataProcessor {
public:
    // Returns sum of even numbers using ranges
    int sum_even(const std::vector<int>& data) const {
        auto view = data | std::views::filter([](int n){ return n % 2 == 0; });
        return std::accumulate(view.begin(), view.end(), 0);
    }
};
```

Unit Test

```
TEST_CASE("Sum of even numbers", "[sum_even]") {
    DataProcessor dp;
    REQUIRE(dp.sum_even({1,2,3,4,5,6}) == 12);
    REQUIRE(dp.sum_even({}) == 0);
    REQUIRE(dp.sum_even({1,3,5}) == 0);
}
```

Highlights:

- Pure, testable, and clear.
- Uses C++20 ranges for readability.
- Easy to extend or refactor without breaking tests.

11.2.5 Key Takeaways for TDD in Modern C++

- Write tests first for new features, then implement logic.
- Design code for testability, not just functionality.
- Use modern C++20/23 features to simplify logic and improve readability.
- Ensure each function or class has minimal dependencies and clear responsibilities.
- Automated tests enable refactoring with confidence, supporting Clean Code principles.

By combining TDD and testable design, C++ developers produce robust, maintainable, and clean code, fully aligned with ISO Core Guidelines and modern professional standards.

11.3 Static Analysis Tools: clang-tidy, cppcheck

Static analysis tools are an essential part of professional C++ development, providing automated checks for code quality, correctness, and adherence to standards before runtime. Tools like clang-tidy and cppcheck help detect potential bugs, enforce modern C++ practices, and maintain clean, maintainable code, particularly when working with C++20 and C++23 features.

11.3.1 The Problem: Code Without Static Analysis

Many projects rely solely on compilation and manual code review, leading to subtle issues like:

- Unused variables

- Potential null dereferences
- Memory mismanagement
- Non-compliance with modern C++ best practices

Bad Code Example

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

int main() {
    int* ptr = nullptr; // Potential null dereference
    std::vector<int> data = {1,2,3};
    std::cout << "First value: " << data[5] << '\n'; // Out-of-bounds
    return 0;
}
```

Issues:

- Raw pointer initialization with nullptr.
- Out-of-bounds access on vector.
- Compiler may not warn about all issues.

11.3.2 Clean Code Example – Using clang-tidy

clang-tidy is a modern C++ linter and static analysis tool integrated with Clang/LLVM. It enforces best practices, identifies bugs, and suggests modern C++20/23 improvements.

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

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

    // Safe access
    if (!data.empty()) {
        std::cout << "First value: " << data.front() << '\n';
    }

    // Prefer smart pointers over raw pointers
    auto ptr = std::make_unique<int>(42);
    std::cout << "Value: " << *ptr << '\n';
}
```

Advantages with clang-tidy:

- Detects out-of-bounds access and null pointer risks.
- Suggests modern smart pointers instead of raw pointers.
- Can enforce ISO C++ Core Guidelines automatically.
- Supports custom checks for C++20/23 features like ranges, std::span, concepts.

11.3.3 Clean Code Example – Using cppcheck

cppcheck is a static analysis tool focusing on bugs and undefined behavior, independent of the compiler. It provides a quick audit for code quality.

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

```
#include <ranges>

int sum_even(const std::vector<int>& data) {
    // Clean modern C++20 code
    auto view = data | std::views::filter([](int n){ return n % 2 == 0; });
    int total = 0;
    for (int n : view) total += n;
    return total;
}

int main() {
    std::vector<int> numbers{1,2,3,4,5,6};
    std::cout << "Sum of even numbers: " << sum_even(numbers) << '\n';
}
```

Advantages with cppcheck:

- Identifies unused variables, memory leaks, null pointer dereferences.
- Detects inefficient or unsafe loops.
- Encourages safe use of modern constructs like ranges, structured bindings, and smart pointers.
- Integrates into CI pipelines for automated enforcement of code quality.

11.3.4 Best Practices for Static Analysis

1. Run tools regularly
    - Integrate clang-tidy or cppcheck into pre-commit hooks or CI/CD pipelines.
2. Use modern C++ checks

- Enable C++20/23-specific rules for ranges, constexpr, and concepts.

3. Fix issues promptly

   - Treat warnings as first-class feedback, not optional suggestions.

4. Customize rules

   - Apply project-specific coding standards, consistent with Clean Code and ISO C++ Core Guidelines.

11.3.5 Key Takeaways

- Static analysis is essential for professional C++ projects.
- Tools like clang-tidy and cppcheck detect bugs, enforce modern C++ standards, and improve maintainability.
- Using these tools helps prevent runtime errors, enforce best practices, and maintain clean, testable, and robust code.
- When combined with unit testing and TDD, static analysis ensures high-quality, production-ready C++20/23 code.

By adopting clang-tidy and cppcheck in your workflow, you ensure your code is not only correct but also modern, safe, and aligned with professional C++ standards.

11.4 Static Analysis Tools: clang-tidy, cppcheck

Chapter 11: Testing and Verification
Booklet: *C++ Clean Code: The Definitive Practical Guide (C++20 & C++23)*
Static analysis tools are an essential part of professional C++ development, providing automated checks for code quality, correctness, and adherence to standards before runtime. Tools like clang-tidy and cppcheck help detect potential bugs, enforce modern C++ practices, and maintain clean, maintainable code, particularly when working with C++20 and C++23 features.

11.4.1 The Problem: Code Without Static Analysis

Many projects rely solely on compilation and manual code review, leading to subtle issues like:

- Unused variables
- Potential null dereferences
- Memory mismanagement
- Non-compliance with modern C++ best practices

Bad Code Example

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

int main() {
    int* ptr = nullptr; // Potential null dereference
    std::vector<int> data = {1,2,3};
```

```
    std::cout << "First value: " << data[5] << '\n'; // Out-of-bounds
    return 0;
}
```

Issues:

- Raw pointer initialization with nullptr.
- Out-of-bounds access on vector.
- Compiler may not warn about all issues.

11.4.2 Clean Code Example – Using clang-tidy

clang-tidy is a modern C++ linter and static analysis tool integrated with Clang/LLVM. It enforces best practices, identifies bugs, and suggests modern C++20/23 improvements.

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

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

    // Safe access
    if (!data.empty()) {
        std::cout << "First value: " << data.front() << '\n';
    }

    // Prefer smart pointers over raw pointers
    auto ptr = std::make_unique<int>(42);
    std::cout << "Value: " << *ptr << '\n';
}
```

Advantages with clang-tidy:

- Detects out-of-bounds access and null pointer risks.
- Suggests modern smart pointers instead of raw pointers.
- Can enforce ISO C++ Core Guidelines automatically.
- Supports custom checks for C++20/23 features like ranges, std::span, concepts.

11.4.3 Clean Code Example – Using cppcheck

cppcheck is a static analysis tool focusing on bugs and undefined behavior, independent of the compiler. It provides a quick audit for code quality.

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

int sum_even(const std::vector<int>& data) {
    // Clean modern C++20 code
    auto view = data | std::views::filter([](int n){ return n % 2 == 0; });
    int total = 0;
    for (int n : view) total += n;
    return total;
}

int main() {
    std::vector<int> numbers{1,2,3,4,5,6};
    std::cout << "Sum of even numbers: " << sum_even(numbers) << '\n';
}
```

Advantages with cppcheck:

- Identifies unused variables, memory leaks, null pointer dereferences.
- Detects inefficient or unsafe loops.
- Encourages safe use of modern constructs like ranges, structured bindings, and smart pointers.
- Integrates into CI pipelines for automated enforcement of code quality.

11.4.4 Best Practices for Static Analysis

1. Run tools regularly
    - Integrate clang-tidy or cppcheck into pre-commit hooks or CI/CD pipelines.
2. Use modern C++ checks
    - Enable C++20/23-specific rules for ranges, constexpr, and concepts.
3. Fix issues promptly
    - Treat warnings as first-class feedback, not optional suggestions.
4. Customize rules
    - Apply project-specific coding standards, consistent with Clean Code and ISO C++ Core Guidelines.

11.4.5 Key Takeaways

- Static analysis is essential for professional C++ projects.
- Tools like clang-tidy and cppcheck detect bugs, enforce modern C++ standards, and improve maintainability.
- Using these tools helps prevent runtime errors, enforce best practices, and maintain clean, testable, and robust code.
- When combined with unit testing and TDD, static analysis ensures high-quality, production-ready C++20/23 code.

By adopting clang-tidy and cppcheck in your workflow, you ensure your code is not only correct but also modern, safe, and aligned with professional C++ standards.

Chapter 12

Designing Clean APIs

12.1 Guidelines for Designing Clean, Maintainable APIs

Designing APIs in modern C++ requires a balance of clarity, safety, expressiveness, and maintainability. A clean API allows other developers to use your code efficiently without confusion, reduces bugs, and ensures forward compatibility as C++20/23 features evolve.

12.1.1 The Problem: Poorly Designed APIs

APIs often become hard to use and error-prone due to inconsistent naming, hidden side effects, tight coupling, or unclear ownership semantics.

Bad API Example

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

class DataManager {
```

```
public:
    std::vector<std::string>& getData() { return data; } // Exposes internal container
    void add(const std::string& s) { data.push_back(s); }
private:
    std::vector<std::string> data;
};

int main() {
    DataManager dm;
    dm.add("hello");
    auto& d = dm.getData();
    d.clear(); // External code can corrupt internal state
}
```

Issues:

- Exposes internal data, breaking encapsulation.
- No control over modification or invariants.
- Hard to maintain and evolve safely.

12.1.2 Clean Code Example – Designing a Safe API

Refactored API using encapsulation, const-correctness, and modern C++20/23 practices:

```
#include <vector>
#include <string>
#include <ranges>

class DataManager {
public:
```

```
    void add(std::string_view value) { data_.push_back(std::string(value)); }

    // Provides read-only access
    auto view() const { return data_ | std::views::all; }

    size_t size() const noexcept { return data_.size(); }

private:
    std::vector<std::string> data_;
};

int main() {
    DataManager dm;
    dm.add("hello");
    dm.add("world");

    for (auto s : dm.view()) {
        std::cout << s << '\n'; // Read-only access
    }
}
```

Improvements:

- Encapsulation: Internal vector not exposed.
- Const-correctness: view() provides read-only access.
- Modern C++20: Uses std::string_view for efficient parameter passing and ranges for clean iteration.
- Safety: External code cannot corrupt internal state.

12.1.3 Core Guidelines for Clean API Design

1. Encapsulate internal state
    - Never expose raw containers or internal pointers.
    - Provide controlled accessors or views.
2. Use consistent, expressive naming
    - Names should clearly indicate purpose, action, or return type.
3. Prefer value semantics and smart pointers
    - Use std::unique_ptr or std::shared_ptr for dynamic ownership.
    - Avoid raw pointers in public interfaces.
4. Use const and constexpr where appropriate
    - Enable compile-time guarantees and immutability.
5. Minimize dependencies
    - Keep API headers lightweight.
    - Forward-declare types where possible to reduce compilation coupling.
6. Provide overloads with modern C++ conveniences
    - Accept std::string_view or std::span instead of std::string or raw arrays for flexibility.

12.1.4 Advanced Example – Modern, Clean API

```
#include <vector>
#include <string>
#include <ranges>

class Logger {
public:
    void log(std::string_view msg) {
        logs_.push_back(std::string(msg));
    }

    auto entries() const { return logs_ | std::views::all; }

private:
    std::vector<std::string> logs_;
};

int main() {
    Logger log;
    log.log("Initializing system");
    log.log("System ready");

    for (auto entry : log.entries()) {
        std::cout << entry << '\n';
    }
}
```

Highlights:

- Clean, minimal public interface.
- Safe and read-only access to internal data.

- Modern C++20/23 usage for efficiency and clarity.
- Future-proof design for extension and testing.

12.1.5 Key Takeaways

- A clean API reduces complexity for users and minimizes bugs.
- Encapsulation, const-correctness, and modern C++ features are essential.
- Prefer views, spans, and string_views for efficient, safe interfaces.
- Clear, expressive naming and minimal dependencies improve maintainability and readability.
- A well-designed API naturally complements Clean Code principles, unit testing, and static analysis.

By following these guidelines, C++ developers can create robust, maintainable, and professional APIs that leverage the full power of C++20/23 while remaining safe, clear, and efficient.

12.2 Compatibility and Extensibility

Designing clean, maintainable APIs is not only about clarity and safety; it also requires forward-thinking for compatibility and extensibility. A robust API should allow future evolution, enable integration with other libraries, and minimize breaking changes while leveraging modern C++20/23 features.

12.2.1 The Problem: Rigid, Non-Extensible APIs

Poorly designed APIs often limit future improvements or force users to rewrite code when extensions are needed. Typical issues include:

- Hard-coded types or fixed container implementations
- No abstraction for polymorphic behavior
- Tight coupling that prevents extension

Bad API Example

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

class FileStorage {
public:
    void addFile(const std::string& filename) {
        files.push_back(filename);
    }

    std::vector<std::string>& getFiles() { return files; } // Direct exposure
private:
    std::vector<std::string> files;
```

```
};

int main() {
    FileStorage storage;
    storage.addFile("config.txt");

    // User must know internal vector type
    auto& f = storage.getFiles();
    f.push_back("data.bin");
}
```

Issues:

- Exposes internal container; any change breaks user code.
- No abstraction to allow storage of different types (e.g., std::filesystem::path).
- Tight coupling prevents extension without modification.

12.2.2 Clean Code Example – Extensible, Compatible API

By applying abstraction, modern C++20/23 types, and concepts, the API becomes extensible, safe, and forward-compatible.

```
#include <vector>
#include <string>
#include <string_view>
#include <ranges>

class FileStorage {
public:
    // Accepts flexible string types
    void addFile(std::string_view filename) {
```

```
        files_.push_back(std::string(filename));
    }

    // Provides read-only view
    auto files() const { return files_ | std::views::all; }

    // Allows future extension with templates or concepts
    template <typename Container>
    void addFiles(const Container& newFiles) {
        for (auto&& f : newFiles) {
            addFile(f);
        }
    }

private:
    std::vector<std::string> files_;
};

int main() {
    FileStorage storage;
    storage.addFile("config.txt");
    storage.addFiles({"data.bin", "log.txt"});

    for (auto f : storage.files()) {
        std::cout << f << '\n';
    }
}
```

Improvements:

- Encapsulation: Internal vector remains private.
- Compatibility: Accepts std::string_view, allowing std::string, literals, or other compatible types.

- Extensibility: Template method addFiles allows adding any iterable container.
- Safe iteration: Uses C++20 ranges to provide read-only views.

12.2.3 Guidelines for Compatibility and Extensibility

1. Abstract implementation details
    - Never expose internal containers or types directly.
    - Use views or iterators to provide access.
2. Use flexible parameter types
    - std::string_view, std::span, or templates increase API adaptability.
3. Support extension without modification
    - Use templated functions, virtual interfaces, or concepts to allow future features.
4. Maintain backward compatibility
    - Avoid breaking changes in public interfaces.
    - Introduce new features via overloads or optional parameters.
5. Leverage modern C++20/23 features
    - std::span, std::string_view, ranges, concepts, and constexpr enable safe, maintainable, and efficient APIs.

12.2.4 Key Takeaways

- Designing APIs for compatibility and extensibility reduces future maintenance costs.
- Proper abstraction, flexible types, and modern C++ features ensure APIs remain safe, robust, and user-friendly.
- Forward-looking design allows your library or module to evolve without breaking existing client code.
- Clean, extensible APIs naturally align with Clean Code principles, static analysis, and unit testing, forming a foundation for professional, modern C++ software.

By following these guidelines, you create C++20/23 APIs that are maintainable, extensible, and future-proof, ensuring long-term usability and safety.

12.3 Examples from the Standard Library

The C++ Standard Library itself is a prime example of well-designed, clean, maintainable, and extensible APIs. Studying its design provides valuable lessons on encapsulation, abstraction, flexibility, and modern C++ practices, particularly in C++20 and C++23.

12.3.1 The Problem: Poorly Designed API Usage

Even when using the standard library, improper handling can produce messy, unsafe, or hard-to-maintain code.

Bad Code Example

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

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

    // Manual iteration and indexing
    for (size_t i = 0; i < data.size(); ++i) {
        if (data[i] % 2 == 0)
            std::cout << data[i] << "\n";
    }

    // Using raw pointers with STL containers
    int* ptr = &data[0];
    for (size_t i = 0; i < data.size(); ++i)
        std::cout << *(ptr + i) << "\n";
}
```

Issues:

- Manual indexing and raw pointers increase chances of errors.
- Code is verbose and less expressive.
- Violates modern C++ principles of safety and readability.

12.3.2 Clean Code Example – Using Modern Standard Library Features

Using ranges, algorithms, and modern iteration patterns simplifies code and aligns with clean API design.

```
#include <vector>
#include <ranges>
#include <algorithm>
#include <iostream>

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

    // Clean, expressive iteration using ranges
    auto even_numbers = data | std::views::filter([](int n){ return n % 2 == 0; });

    for (int n : even_numbers) {
        std::cout << n << "\n";
    }

    // Using standard algorithms
    std::for_each(data.begin(), data.end(), [](int n){
        if (n % 2 != 0)
            std::cout << n << "\n";
    });
}
```

Improvements:

- Ranges: Expressive, safe, and composable filtering.
- Algorithms: Avoid manual loops and indices, reduce error potential.
- Readability: Code expresses what is done, not how.
- Safety: No raw pointers, avoids undefined behavior.

12.3.3 Lessons from Standard Library API Design

1. Use encapsulation and iterators
    - Access container elements through iterators or ranges, not raw pointers.
2. Prefer algorithms over manual loops
    - std::for_each, std::ranges::filter, and std::ranges::transform simplify operations.
3. Flexible and generic interfaces
    - STL functions work with a wide range of containers and iterator types.
4. Consistent naming and behavior
    - Function names like push_back, emplace_back, sort, and find are self-explanatory and predictable.
5. Leverage modern C++20/23 features
    - Concepts, ranges, views, and std::span improve safety, readability, and expressiveness.

12.3.4 Advanced Example – Combining Algorithms and Views

```
#include <vector>
#include <ranges>
#include <iostream>
#include <numeric>

int main() {
    std::vector<int> values = {1,2,3,4,5,6};

    // Filter even, transform, and compute sum
    auto even_squares = values
                        | std::views::filter([](int n){ return n % 2 == 0; })
                        | std::views::transform([](int n){ return n*n; });

    int sum = std::accumulate(even_squares.begin(), even_squares.end(), 0);
    std::cout << "Sum of squares of even numbers: " << sum << "\n";
}
```

Highlights:

- Composability: Ranges allow chaining multiple transformations.
- Readability: Clearly expresses the intent: filter, transform, sum.
- Safety: No manual indexing or temporary raw arrays.
- Modern C++20/23: Full utilization of ranges and algorithms for clean, maintainable code.

12.3.5 Key Takeaways

- The Standard Library is a model of clean API design, emphasizing safety, expressiveness, and flexibility.

- Modern C++20/23 features like ranges, views, and concepts extend its power while maintaining clean interfaces.
- Learning from STL design principles helps design your own APIs that are robust, extensible, and user-friendly.
- Emphasize expressive, safe, and composable operations to create maintainable C++ software.

By following STL-inspired patterns, your APIs naturally embody Clean Code principles, remain future-proof, and leverage the full potential of C++20/23.

Appendices – Selected ISO Core Guidelines

Appendix A: Practical Rules with Examples

The ISO C++ Core Guidelines, initiated by Bjarne Stroustrup and Herb Sutter, provide a practical foundation for writing clean, safe, and maintainable C++ code. This appendix summarizes key rules with realistic examples, demonstrating how to apply them in modern C++20/23.

1. Rule: Prefer auto to explicit type for readability

   Bad Code Example

```
std::vector<int>::iterator it = myVector.begin();
for (; it != myVector.end(); ++it) {
    std::cout << *it << "\n";
}
```

   Clean Code Example

```
for (auto it = myVector.begin(); it != myVector.end(); ++it) {
    std::cout << *it << "\n";
}
```

Modern Improvement (C++20 Ranges):

```
for (auto v : myVector) {
    std::cout << v << "\n";
}
```

Lesson: auto reduces verbosity and improves readability without losing type safety.

2. Rule: Prefer nullptr over 0 or NULL

Bad Code Example

```
int* ptr = 0;
if (ptr == NULL) {
    std::cout << "Pointer is null\n";
}
```

Clean Code Example

```
int* ptr = nullptr;
if (ptr == nullptr) {
    std::cout << "Pointer is null\n";
}
```

Lesson: nullptr is type-safe and eliminates ambiguity in pointer comparisons.

3. Rule: Use constexpr for compile-time constants

Bad Code Example

```
#define PI 3.14159
double area(double r) { return PI * r * r; }
```

Clean Code Example

```
constexpr double PI = 3.14159;
double area(double r) { return PI * r * r; }
```

Lesson: constexpr provides type safety, scoping, and compile-time evaluation, unlike preprocessor macros.

4. Rule: Use RAII for resource management

Bad Code Example

```
FILE* f = fopen("file.txt", "r");
// ... use f ...
fclose(f); // Forgetting this can leak resources
```

Clean Code Example

```
#include <fstream>

std::ifstream file("file.txt");
if (file) {
    std::string line;
    while (std::getline(file, line)) {
        std::cout << line << "\n";
    }
} // File automatically closed when out of scope
```

Lesson: RAII ensures automatic and safe resource cleanup.

5. Rule: Prefer std::unique_ptr over raw pointers

Bad Code Example

```
Widget* w = new Widget();
// ... use w ...
delete w; // Manual memory management
```

Clean Code Example

```
#include <memory>

auto w = std::make_unique<Widget>();
// No need to delete, automatically destroyed
```

Lesson: Smart pointers prevent memory leaks and clearly indicate ownership.

6. Rule: Avoid magic numbers and use named constants

   Bad Code Example

   ```
   double total = price * 1.07; // What is 1.07?
   ```

   Clean Code Example

   ```
   constexpr double TAX_RATE = 0.07;
   double total = price * (1.0 + TAX_RATE);
   ```

   Lesson: Named constants improve readability, maintainability, and self-documentation.

7. Rule: Use std::span for safe array or buffer access (C++20)

   Bad Code Example

```
void printArray(int* arr, size_t size) {
    for (size_t i = 0; i < size; ++i)
        std::cout << arr[i] << "\n";
}
```

Clean Code Example

```
#include <span>
#include <vector>

void printArray(std::span<int> arr) {
    for (auto v : arr)
        std::cout << v << "\n";
}

std::vector<int> data = {1,2,3};
printArray(data);
```

Lesson: std::span provides bounds-checked, safe, and generic access to arrays or containers.

8. Rule: Prefer enum class over plain enum

Bad Code Example

```
enum Color { Red, Green, Blue };
int c = Red; // Implicit conversion to int
```

Clean Code Example

```
enum class Color { Red, Green, Blue };
Color c = Color::Red; // Strongly typed, scoped
```

Lesson: Scoped enums prevent name collisions and accidental conversions.

9. Rule: Use noexcept where applicable

   Bad Code Example

```
void process() {
    // might throw exceptions
}
```

   Clean Code Example

```
void process() noexcept {
    // guarantees no exceptions
}
```

   Lesson: Declaring noexcept improves performance, optimizations, and conveys intent.

10. Rule: Favor algorithms and ranges over manual loops

    Bad Code Example

```
std::vector<int> v = {1,2,3,4,5};
int sum = 0;
for (size_t i = 0; i < v.size(); ++i) sum += v[i];
```

    Clean Code Example

```
#include <numeric>
int sum = std::accumulate(v.begin(), v.end(), 0);
```

    Modern C++20 Example:

```
#include <ranges>
int sum = std::ranges::accumulate(v, 0);
```

Lesson: Algorithms and ranges reduce boilerplate, errors, and improve readability.

Key Takeaways

- The ISO Core Guidelines promote safety, clarity, and maintainability in modern C++.
- Encapsulation, RAII, smart pointers, constexpr, ranges, and concepts are central for clean C++20/23 code.
- Avoid manual memory management, magic numbers, raw loops, and unsafe pointers.
- Applying these practical rules consistently results in robust, maintainable, and future-proof software.

This appendix demonstrates how small, rule-based improvements can transform messy code into clean, professional C++, adhering to the ISO Core Guidelines.

Appendix B: Interpretation and Practical Application of Each Guideline

The ISO C++ Core Guidelines are extensive, but their practical value lies in correct interpretation and disciplined application. This appendix demonstrates how to translate the guidelines into real-world clean C++20/23 code, emphasizing safety, readability, maintainability, and modern features.

1. Guideline: Use auto to avoid verbose types

   Interpretation

   - Use auto where the type is obvious from context or too verbose to write manually.
   - Improves readability and reduces maintenance cost if the type changes.

   Bad Code Example

```
std::vector<std::pair<std::string,int>>::iterator it = myVector.begin();
for (; it != myVector.end(); ++it) {
    std::cout << it->first << ": " << it->second << "\n";
}
```

   Clean Code Example

```
for (auto it = myVector.begin(); it != myVector.end(); ++it) {
    std::cout << it->first << ": " << it->second << "\n";
}
```

```
// Modern C++20 range-based
for (auto [name, value] : myVector) {
    std::cout << name << ": " << value << "\n";
}
```

Practical application: Use structured bindings with auto for clarity and conciseness.

2. Guideline: Prefer RAII over manual resource management

   Interpretation

   - Ensure that resources are automatically released when they go out of scope.
   - Avoid manual calls to delete, fclose, or similar.

   Bad Code Example

```
FILE* file = fopen("data.txt", "r");
// use file
fclose(file);
```

   Clean Code Example

```
#include <fstream>

std::ifstream file("data.txt");
if (file) {
    std::string line;
    while (std::getline(file, line)) {
        std::cout << line << "\n";
    }
} // automatic cleanup
```

Practical application: RAII ensures exception-safe and leak-free code.

3. Guideline: Prefer std::unique_ptr and std::shared_ptr to raw pointers

   Interpretation

   - Use ownership semantics to indicate who is responsible for freeing resources.
   - Avoid raw pointers whenever possible.

   Bad Code Example

```
Widget* w = new Widget();
// use w
delete w;
```

   Clean Code Example

```
auto w = std::make_unique<Widget>();
// automatic cleanup
```

   Practical application: Smart pointers enforce memory safety and improve API clarity.

4. Guideline: Use constexpr and const wherever possible

   Interpretation

   - constexpr: evaluated at compile-time, improving performance.
   - const: communicates immutability and prevents accidental modification.

   Bad Code Example

```
#define MAX_SIZE 100
int array[MAX_SIZE];
```

Clean Code Example

```
constexpr int MAX_SIZE = 100;
int array[MAX_SIZE];
```

Practical application: Always prefer typed, scoped constants to macros.

5. Guideline: Use noexcept when functions cannot throw

Interpretation

- Declaring noexcept communicates intent and enables compiler optimizations.
- Reduces runtime overhead in exception-handling scenarios.

Bad Code Example

```
void processData() {
    // no guarantees
}
```

Clean Code Example

```
void processData() noexcept {
    // guarantees no exceptions
}
```

Practical application: Improves safety, predictability, and performance.

6. Guideline: Prefer algorithms and ranges over manual loops

   Interpretation

   - Replace manual iteration with STL algorithms or ranges for clarity and correctness.

   Bad Code Example

```
int sum = 0;
for (size_t i = 0; i < v.size(); ++i)
    sum += v[i];
```

   Clean Code Example

```
#include <numeric>
int sum = std::accumulate(v.begin(), v.end(), 0);

// C++20 ranges
#include <ranges>
int sum2 = std::ranges::accumulate(v, 0);
```

   Practical application: Reduces boilerplate, errors, and improves expressiveness.

7. Guideline: Use enum class instead of unscoped enums

   Interpretation

   - Scoped enums prevent name collisions and implicit conversions.

   Bad Code Example

```
enum Color { Red, Green, Blue };
Color c = Red; // implicit conversion allowed
```

Clean Code Example

```
enum class Color { Red, Green, Blue };
Color c = Color::Red; // strongly typed
```

Practical application: Scoped enums are safer and make intent explicit.

8. Guideline: Avoid magic numbers; use named constants

   Interpretation

   - Named constants document intent, reduce errors, and improve maintainability.

   Bad Code Example

   ```
   double total = price * 1.08;
   ```

   Clean Code Example

   ```
   constexpr double TAX_RATE = 0.08;
   double total = price * (1.0 + TAX_RATE);
   ```

   Practical application: Improves readability and maintainability.

9. Guideline: Prefer std::span for array-like access (C++20)

   Interpretation

- std::span provides bounds-checked, safe access to arrays or containers.
- Avoid raw pointers for function parameters.

Bad Code Example

```
void printArray(int* arr, size_t size) {
    for (size_t i = 0; i < size; ++i)
        std::cout << arr[i] << "\n";
}
```

Clean Code Example

```
#include <span>

void printArray(std::span<int> arr) {
    for (auto v : arr)
        std::cout << v << "\n";
}
```

Practical application: Simplifies interfaces and improves runtime safety.

Key Takeaways

- Interpretation converts abstract rules into practical coding habits.
- Applying guidelines consistently results in robust, safe, maintainable, and modern C++ code.
- C++20/23 features like ranges, std::span, concepts, constexpr, smart pointers, and structured bindings make following guidelines easier and more expressive.

- Clean code is achieved not just by rules, but by understanding their intent and context.

By interpreting and applying ISO Core Guidelines pragmatically, your code will be reliable, maintainable, and aligned with professional C++ standards.

Conclusion

Summary of Best Practices

In modern C++20/23 development, writing clean, maintainable, and safe code is not optional—it is essential for long-term project success. This section summarizes the core best practices you should apply consistently.

1. Prioritize Readability and Simplicity

   Bad Code Example

   ```
   int f(int a,int b){return a*b+a-b;}
   ```

   Clean Code Example

   ```
   int computeAdjustedProduct(int x, int y) {
       return (x * y) + x - y;
   }
   ```

   Takeaway:

   - Use descriptive names.
   - Write short, focused functions.

- Avoid clever but unreadable expressions.

2. Use Modern C++ Features Wisely

   Bad Code Example

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

   Clean Code Example (C++20)

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

std::vector<int> arr = {1,2,3,4,5};
for (auto v : arr | std::views::all) {
    std::cout << v << "\n";
}
```

   Takeaway:

   - Prefer STL algorithms, ranges, and structured bindings.
   - Reduce manual loops and verbose code.
   - Leverage std::span, std::format, smart pointers, and concepts.

3. Apply ISO Core Guidelines

   Key Guidelines:

   - RAII for resource management.

- unique_ptr and shared_ptr for memory safety.
- constexpr and const for immutability.
- Scoped enums (enum class) for type safety.
- noexcept where functions cannot throw.
- Use auto for readability.

Bad Code Example

```
Widget* w = new Widget();
doWork(w);
delete w;
```

Clean Code Example

```
auto w = std::make_unique<Widget>();
doWork(w);
// Automatic cleanup, exception-safe
```

Takeaway:
Following these rules ensures safety, maintainability, and clarity.

4. Organize Files and Projects Clearly

Bad Code Example

```
#include "all_in_one.h"
```

Clean Code Example

```
#include "widget.h"
#include "utils.h"
#include <vector>
```

Takeaway:

- Minimal, clear includes.
- Well-structured file and folder hierarchy.
- Modern CMake project setup for maintainable builds.

5. Test and Verify Thoroughly

Bad Code Example

```
void doWork() {
    // No tests
}
```

Clean Code Example

```
#include <catch2/catch.hpp>

TEST_CASE("doWork behaves correctly") {
    REQUIRE(doWork(5) == expectedValue);
}
```

Takeaway:

- Write unit tests.
- Follow TDD principles.
- Use static analysis tools like clang-tidy and cppcheck.

6. Maintain API Clarity and Extensibility

   Bad Code Example

```
class Data {
    int x;
    int y;
};
```

   Clean Code Example

```
class Point {
public:
    Point(int x, int y) noexcept : x_(x), y_(y) {}
    int getX() const noexcept { return x_; }
    int getY() const noexcept { return y_; }
private:
    int x_;
    int y_;
};
```

   Takeaway:

   - Design APIs that are clear, maintainable, and extendable.
   - Encapsulate implementation details.
   - Document preconditions, postconditions, and exceptions.

7. Embrace Clean Concurrency

   Bad Code Example

```
int counter = 0;
std::thread t1([&]{ counter++; });
std::thread t2([&]{ counter++; });
t1.join(); t2.join();
```

Clean Code Example

```
#include <atomic>
#include <thread>

std::atomic<int> counter{0};
std::thread t1([&]{ counter.fetch_add(1); });
std::thread t2([&]{ counter.fetch_add(1); });
t1.join(); t2.join();
```

Takeaway:

- Prefer std::atomic, mutex, and parallel STL algorithms.
- Avoid data races and undefined behavior.

Key Takeaways

1. Readable code > clever code.
2. Leverage modern C++20/23 features for clarity, safety, and performance.
3. Follow ISO Core Guidelines to standardize and secure code.
4. Organize projects logically, with minimal dependencies.
5. Test early and often, including unit tests and static analysis.
6. Design APIs and classes cleanly for extensibility.

7. Write safe concurrent code using standard library facilities.

By adhering to these best practices, your C++20/23 projects become robust, maintainable, and professional-grade, demonstrating mastery of modern clean code principles.

Tips for Developing a Daily Clean Coding Habit

Writing clean C++ code consistently requires more than knowledge—it requires discipline and practice. Cultivating a daily clean coding habit ensures that modern features, ISO guidelines, and professional standards become second nature.

1. Always Start with Readability

   Bad Code Example

   ```
   int f(int x,int y){return x*y+x-y;}
   ```

   Clean Code Example

   ```
   int computeAdjustedProduct(int a, int b) {
       return (a * b) + a - b;
   }
   ```

   Tip: Before writing any function, ask yourself if another developer can understand it immediately. Clarity first, optimization later.

2. Write Small, Focused Functions Every Day

   Bad Code Example

   ```
   void processData() {
       readFile();
       parseData();
       computeResults();
       saveResults();
   }
   ```

Clean Code Example

```
void readFile();
void parseData();
void computeResults();
void saveResults();

void processData() {
    readFile();
    parseData();
    computeResults();
    saveResults();
}
```

Tip: Break tasks into single-responsibility functions. This practice should become habitual for every project.

3. Use Modern C++ Features Regularly

Bad Code Example

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

Clean Code Example

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

std::vector<int> arr = {1,2,3,4,5};
for (auto v : arr | std::views::all) {
    std::cout << v << "\n";
}
```

Tip: Make it a daily habit to replace manual loops with ranges, prefer STL algorithms, and use structured bindings.

4. Apply ISO Core Guidelines in Every Task

   Bad Code Example

```
Widget* w = new Widget();
// do something
delete w;
```

   Clean Code Example

```
auto w = std::make_unique<Widget>();
// automatic cleanup, exception-safe
```

   Tip: Enforce RAII, smart pointers, constexpr, const, noexcept daily, even for small utility functions.

5. Review and Refactor Frequently

   Bad Code Example

```
int calculate(int x,int y){return x*y+x/y;}
```

   Clean Code Example

```
int calculateProductAndQuotient(int x, int y) {
    if(y == 0) throw std::invalid_argument("y cannot be zero");
    return (x * y) + (x / y);
}
```

Tip: Allocate time at the end of each coding session to review and refactor. Make it a non-negotiable habit.

6. Test Every Change

   Bad Code Example

```
void doWork() {
    // untested logic
}
```

   Clean Code Example

```
#include <catch2/catch.hpp>

TEST_CASE("doWork behaves correctly") {
    REQUIRE(doWork(5) == expectedValue);
}
```

   Tip: Writing unit tests and practicing TDD should be part of daily coding, even for trivial functions.

7. Keep Learning and Updating Practices

   - Stay current with C++20/23 features: std::format, std::span, concepts, coroutines.
   - Integrate static analysis tools (clang-tidy, cppcheck) into your workflow.
   - Follow modern project structure practices: proper namespace use, minimal #include, and CMake best practices.

   Tip: Dedicate a portion of daily coding or review time to modern features and new guidelines.

8. Key Habits to Reinforce Daily

    1. Write small, expressive functions.
    2. Use smart pointers and RAII consistently.
    3. Prefer STL algorithms, ranges, and structured bindings.
    4. Apply ISO guidelines in all tasks.
    5. Refactor and review every session.
    6. Test all new functionality.
    7. Learn and apply new C++20/23 features.

    By integrating these habits into your daily workflow, clean code becomes automatic, not just aspirational. Over time, your projects will consistently demonstrate professionalism, safety, readability, and maintainability.

Resources and References for Further Learning

Mastering clean C++ coding is a continuous journey. While this booklet provides a practical foundation, ongoing learning from authoritative resources ensures that your skills remain sharp and aligned with modern C++20/23 standards.

1. ISO Core Guidelines

   The ISO C++ Core Guidelines are essential for safe, maintainable, and clean C++ code. They cover topics such as:

   - RAII, smart pointers, and resource management
   - Const-correctness and constexpr usage
   - Error handling with noexcept
   - Safe concurrency practices
   - Designing clean interfaces

   Bad Code Example

   ```
   Widget* w = new Widget();
   // multiple operations
   delete w; // prone to memory leaks or exceptions
   ```

   Clean Code Example

   ```
   auto w = std::make_unique<Widget>();
   // automatic cleanup ensures exception safety
   ```

   Tip: Regularly review the ISO Guidelines when designing classes, functions, or templates.

2. Modern C++ Literature

   Books and guides by recognized authorities provide deeper insights into practical clean coding in modern C++:

   - Bjarne Stroustrup: Emphasizes the principles of safety, efficiency, and maintainability.
   - Herb Sutter: Focuses on modern C++ best practices, concurrency, and generic programming.
   - Scott Meyers: Offers practical advice for writing effective, maintainable C++ code.

   Bad Code Example

```
void compute(int a,int b,int c){
    int r=a*b-c;
    std::cout<<r;
}
```

   Clean Code Example

```
void printAdjustedProduct(int x, int y, int offset) {
    int result = (x * y) - offset;
    std::cout << result << "\n";
}
```

   Tip: Reading authoritative C++ texts consistently helps internalize modern idioms and patterns.

3. Online Documentation and Compiler References

- cppreference.com: Comprehensive reference for STL, language features, and C++20/23 additions.
- Compiler documentation: GCC, Clang, and MSVC provide feature-specific examples and warnings for safe code.

Bad Code Example

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

Clean Code Example

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

std::vector<int> arr = {1,2,3,4,5};
for(auto value : arr | std::views::all) {
    std::cout << value << "\n";
}
```

Tip: Reference up-to-date documentation for STL, ranges, std::format, std::span, coroutines, and concepts to leverage modern features properly.

4. Community and Discussion Forums

Engaging with professional communities helps validate clean coding practices and stay informed on modern techniques:

- ISO C++ mailing lists

- Stack Overflow and dedicated C++ forums
- GitHub projects: Analyze how open-source projects implement C++20/23 features with clean code.

Bad Code Example

```
int counter=0;
std::thread t1([&]{counter++;});
std::thread t2([&]{counter++;});
t1.join();t2.join();
```

Clean Code Example

```
#include <atomic>
#include <thread>

std::atomic<int> counter{0};
std::thread t1([&]{ counter.fetch_add(1); });
std::thread t2([&]{ counter.fetch_add(1); });
t1.join();
t2.join();
```

Tip: Observe how professional developers handle concurrency, memory management, and API design.

5. Continuous Practice and Project Work

- Implement personal or open-source projects using modern C++20/23.
- Apply unit testing, TDD, and static analysis daily.
- Refactor code based on the ISO Core Guidelines and community feedback.

Tip: Practical application reinforces clean coding habits far more effectively than theory alone.

Summary

To advance your expertise in clean C++20/23 coding:

1. Study the ISO Core Guidelines regularly.
2. Follow authoritative books by Stroustrup, Sutter, and Meyers.
3. Reference official documentation and compiler guides.
4. Engage with professional communities and analyze real-world projects.
5. Apply concepts daily in practical projects with testing and refactoring.

By combining reading, observation, and practice, your C++ coding will evolve to consistently demonstrate clarity, maintainability, and professional-grade quality.

References

1. ISO C++ Core Guidelines
2. The C++ Programming Language – Bjarne Stroustrup
3. Programming: Principles and Practice Using C++ – Bjarne Stroustrup
4. Effective Modern C++ – Scott Meyers
5. More Effective C++ – Scott Meyers
6. Exceptional C++ / More Exceptional C++ – Herb Sutter
7. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices – Herb Sutter & Andrei Alexandrescu
8. Modern C++ Design – Andrei Alexandrescu
9. C++ Templates: The Complete Guide – David Vandevoorde & Nicolai M. Josuttis
10. Professional C++ – Marc Gregoire
11. C++ Concurrency in Action – Anthony Williams
12. Clean Code in C++ – Stephan Roth
13. The C++ Standard Library: A Tutorial and Reference – Nicolai M. Josuttis