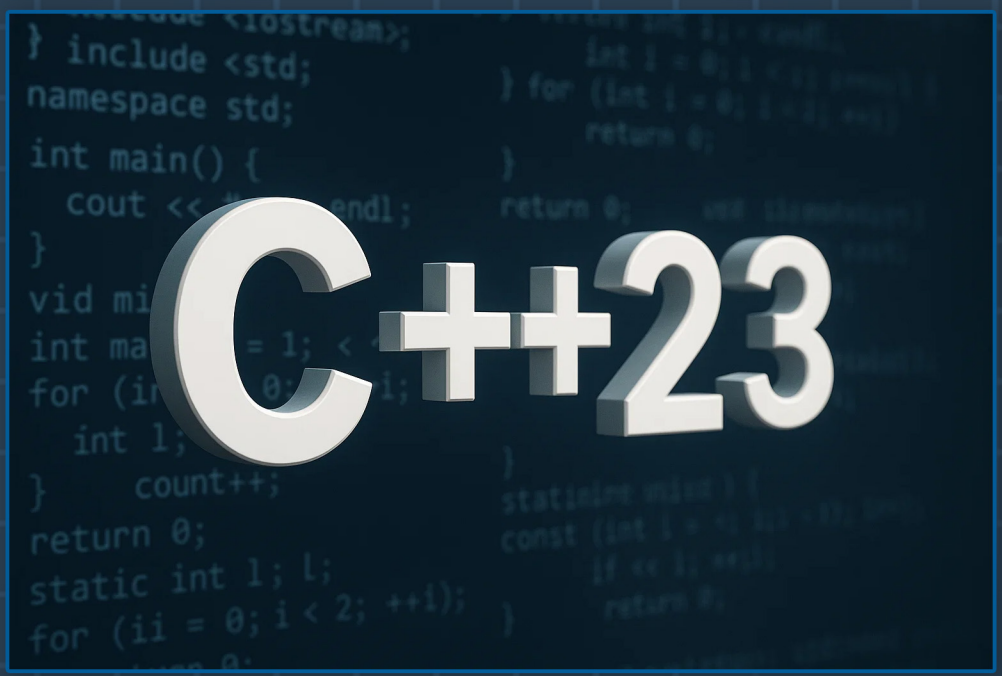
https://simplifyCpp.org

DRAFT

Mastering Modern C++23

A Complete Guide to the Latest Standard



Prepared by Ayman Alheraki

First Edition

Mastering Modern C++23
A Complete Guide to the Latest Standard

Prepared by Ayman Alheraki

simplifycpp.org

August 2025

Contents

Author's Introduction

C++ is evolving, and C++23 is a game-changer.

Mastering Modern C++23: A Complete Guide to the Latest Standard puts the power of the latest C++ updates directly in your hands. From fundamental improvements to cutting-edge features, this book gives you the knowledge and practical skills to write faster, safer, and more efficient code.
Getting ahead with C++23 before C++26 arrives gives you a unique advantage: a deep understanding of today's enhancements and the readiness to leverage tomorrow's innovations the moment they are released. Covering almost every aspect of C++23, this guide ensures your learning is effective, practical, and forward-looking, positioning you at the forefront of modern C++ programming.

`Ayman Alheraki`

Preface

Why C++23 Matters

C++ has always been a language of balance: it provides the **raw efficiency of low-level programming** while offering **high-level abstractions** that enable building complex, reliable systems. Since its inception in the early 1980s, the language has evolved through steady international standardization, adapting to the changing needs of both industry and academia. Every new standard revision introduces features that respond to real-world programming challenges while preserving the performance guarantees and backward compatibility that define C++.

The **C++23 standard**, formally completed in 2023, continues this tradition of evolutionary refinement. Unlike C++11 and C++20—both considered revolutionary releases—C++23 is more **incremental and consolidating** in nature. It addresses gaps left by previous standards, streamlines the developer experience, and improves safety, consistency, and expressiveness. In essence, C++23 is not about changing the language dramatically but about **maturing it into a more coherent and practical tool for modern software development**.

A Language That Powers the World

C++ remains the backbone of critical systems worldwide. It underpins **operating systems, game engines, embedded systems, financial infrastructure, web browsers, and high-performance computing frameworks**. Organizations continue to invest in C++ because of its unmatched combination of **efficiency, portability, and ecosystem stability**. The arrival of C++23 ensures that this investment remains future-proof, providing developers with tools that make code safer, clearer, and easier to maintain without sacrificing speed or flexibility.

Bridging Productivity and Control

One of the defining themes of C++23 is its focus on **developer productivity without compromising control**. For decades, programmers faced trade-offs: higher abstraction often meant slower code, while efficient code demanded more effort and complexity. With features like `std::expected` for error handling, `std::print` for straightforward output, improved ranges adaptors, and expanded `constexpr` capabilities, C++23 reduces boilerplate and eliminates common pitfalls while still allowing low-level optimizations when necessary.

Strengthening Compile-Time Programming

Another key direction in C++23 is the continued empowerment of **compile-time programming**. By broadening what can be expressed in `constexpr` contexts and simplifying template metaprogramming, developers can write more efficient and safer code that the compiler can evaluate early. This shift not only improves performance but also catches errors before the program ever runs, a major step toward robust and maintainable systems.

Text and Unicode in a Globalized World

As software becomes increasingly global, **text encoding and Unicode handling** are central to program correctness. C++23 standardizes UTF-8 as a portable source file encoding, provides named universal character escapes, and refines character literal semantics. These changes modernize the way C++ interacts with international text and bring clarity to an area that has historically been confusing and error-prone.

Usability and Everyday Convenience

For everyday development, C++23 introduces numerous conveniences that collectively save countless hours:

- Multidimensional subscript operators for natural mathematical notation.
- Range adaptors like `zip`, `chunk`, and `enumerate` to simplify data processing.
- Flat associative containers (`flat_map`, `flat_set`) for efficient memory layouts.
- A standardized stacktrace facility for debugging.
- Streamlined syntax for lambdas, deducing `this`, and contextual conversions.

These are not headline-grabbing features on their own, but together they represent a strong commitment to **making C++ more intuitive without breaking its philosophy**.

A Foundation for the Future

Finally, C++23 lays the groundwork for **C++26 and beyond**. With each revision, the standard library grows more expressive, more aligned with real-world needs, and more consistent with modern programming paradigms. By adopting C++23 today,

developers place themselves in an excellent position to smoothly transition to future standards while writing code that benefits from the best of both performance and clarity.

Summary

C++23 matters because it demonstrates the language's **resilience and adaptability**. It strengthens core areas such as **error handling, compile-time programming, Unicode support, and ranges** while offering a collection of practical tools that reduce friction in daily development. It is not a radical reinvention, but rather a thoughtful refinement that makes C++ a stronger, safer, and more productive language for the next decade of software development.
This book is written to help you understand and apply these changes effectively. By the end of it, you will not only know what has changed in C++23 but also **why these changes exist and how to use them to write better software**.

About This Book

This book has been written to serve as a **comprehensive and practical guide to C++23**, the latest revision of the C++ programming language standard. Its purpose is to provide readers with both a **deep technical understanding** of the new features introduced in C++23 and **clear, example-driven explanations** that make these features accessible in real-world development.

Purpose of the Book

The central aim of this book is twofold:

1. **Educational**: To teach the concepts, rationale, and syntax of every major

change in C++23, ensuring readers understand not just *what* changed, but *why* it changed and *how* to use it effectively.

2. **Practical**: To act as a **reference guide** that developers can consult when working on modern C++ projects, offering practical examples, migration strategies, and best practices for integrating C++23 features into existing codebases.

This dual role makes the book suitable both as a **learning resource for individuals** and as a **desk reference for professionals** in active development environments.

Intended Audience

This book is designed for a wide range of readers who already possess a background in C++ programming:

- **Intermediate developers** who are comfortable with C++11, C++14, or C++17 and want to expand their knowledge into the modern features of C++23.
- **Advanced developers and system programmers** who need to understand the technical implications of C++23 features for high-performance or large-scale projects.
- **Educators and trainers** seeking a structured and authoritative resource for teaching the latest standard.
- **Professionals working with critical systems** such as embedded software, financial platforms, operating systems, and scientific computing, where C++ remains an essential language.

This book is **not** intended for absolute beginners with no prior programming experience. Readers are expected to be familiar with the fundamentals of C++ syntax, object-oriented programming, and templates.

Scope and Coverage

The coverage of this book spans every area of the C++23 standard relevant to day-to-day and professional development:

- **Language Features**: Detailed explanations of new syntax, keywords, attributes, constexpr extensions, and rules that shape how C++ code is written.
- **Standard Library Features**: In-depth exploration of new headers, new containers, algorithms, ranges, text processing tools, and diagnostic facilities.
- **Removed and Deprecated Features**: Guidance on features removed from the standard and the recommended migration paths.
- **Best Practices**: Insights into using C++23 in a modern development environment, with an emphasis on safety, clarity, and efficiency.
- **Practical Migration**: Strategies for adapting legacy C++ code to the new standard without introducing instability.

Where relevant, the book also highlights the **design rationale** behind features, providing context from the standardization process to explain why certain decisions were made.

Structure of the Book

The book is divided into five major parts:

1. **Foundations of C++23**
   Provides background on the evolution of C++ standards, compiler support, and a high-level overview of what C++23 introduces.

2. **New Language Features**
   Explains every new addition to the core language in C++23, from deducing `this` and multidimensional subscripts to expanded `constexpr` capabilities.

3. **The Standard Library in C++23**
   Covers new library components such as `std::expected`, `std::print`, ranges enhancements, new containers (`flat_map`, `mdspan`), and diagnostic tools.

4. **Obsolete, Deprecated, and Reverted Features**
   Discusses what has been removed, what is marked for deprecation, and what has been restored, with notes on how to update existing code.

5. **Practical Applications of C++23**
   Provides applied knowledge for migrating projects, writing robust new code, and preparing for the future direction of the C++ language.

Finally, the book concludes with a series of **appendices** that supply quick references, compiler support notes, feature test macros, and compatibility information with the C23 standard.

How to Use This Book

This book has been written to be flexible in use:

- **Sequential Reading**: Readers may progress from start to finish, treating the book as a structured learning path.

- **Reference Reading**: Professionals may consult specific chapters or sections when they need immediate clarity on a particular feature.

Code examples are included throughout the text. These examples are designed to be concise, compilable, and illustrative of both correct usage and potential pitfalls. Readers are encouraged to compile and experiment with these examples to reinforce understanding.

The Role of C++23 Today

By engaging with this book, readers will not only gain an understanding of the **technical specifications of C++23**, but also appreciate its role in the larger **software development ecosystem**. C++23 is not merely a collection of new features; it represents the ongoing evolution of a language that continues to balance **performance, portability, and expressiveness**. Understanding C++23 today is an investment in both present productivity and future readiness as the language moves toward C++26 and beyond.

How to Use This Guide

This book has been designed to be both a **learning resource** and a **long-term reference** for the C++23 standard. Because readers will come to it with different levels of experience, programming backgrounds, and professional goals, it is important to outline how the book can be approached effectively. This section provides guidance on reading strategies, use of examples, and methods of applying the material in practice.

Reading Strategies

There are two primary ways to navigate this book:

1. **Sequential Reading**

    - If you are seeking a **complete education in C++23**, you may wish to read the book in order, chapter by chapter.
    - The structure has been designed to gradually move from background and context (the history and evolution of C++ standards) to **new language features**, then to **new library additions**, before discussing **deprecated features** and **practical applications**.
    - This linear approach ensures you acquire both the **conceptual foundation** and the **practical mastery** of C++23.

2. **Targeted Reference**

    - For working professionals, the book may serve best as a **reference manual**.
    - Each chapter is self-contained, with feature explanations, examples, and summaries that can be consulted independently.
    - For example, if you only need to understand `std::expected`, or the new `std::print` functionality, you can go directly to those chapters without reading preceding material.

Both approaches are valid, and the book is written to accommodate both learners and experts alike.

Code Examples

Practical, compilable code examples are included throughout the book. These examples serve several purposes:

- **Illustration of Features**: Each example demonstrates the syntax and semantics of a new language or library feature.
- **Comparison with Earlier Standards**: Where applicable, examples show how a task was done in older standards (C++11/14/17/20) and how it improves with C++23.
- **Pitfalls and Misuse**: Some examples are intentionally written to show common mistakes or misunderstandings, followed by corrected solutions.

All examples have been tested with modern compilers supporting C++23. While compiler implementations may differ in the speed of adoption, the examples are written in compliance with the official ISO standard to ensure portability.
Readers are encouraged to **compile, run, and modify** these examples.
Experimentation is one of the most effective ways to internalize new features.

Level of Assumed Knowledge

This book assumes that the reader has:

- A solid understanding of **core C++ syntax** (functions, classes, templates, inheritance, pointers, references).
- Familiarity with **C++11 and beyond** (basic knowledge of move semantics, smart pointers, lambdas, `constexpr`).
- Experience writing and compiling small to medium-sized C++ programs.

The book does not assume prior exposure to C++20 or C++23. Instead, it is designed to **bridge the gap** between older versions of C++ and the latest standard, bringing readers up to speed.

Suggested Workflow for Readers

To gain the most from this guide, the following workflow is recommended:

1. **Read the Conceptual Overview**
   Begin with Part I (Foundations of C++23) to understand the context and philosophy of the new standard.

2. **Work Through Examples Actively**
   Do not simply read the code samples. Compile them, experiment with modifications, and observe how the compiler responds.

3. **Compare with Legacy Approaches**
   When examples contrast older techniques with C++23 improvements, take time to implement both versions. This will highlight the efficiency, clarity, or safety improvements provided by the new standard.

4. **Apply to Real Projects**
   Take selected features—such as `std::expected`, range adaptors, or `constexpr` extensions—and apply them in small experiments within your existing projects. Incremental adoption ensures smoother learning.

5. **Consult the Book as a Reference**
   When working professionally, return to the book as needed. Each chapter ends with a **summary and best practices** section designed for quick reference.

Compiler and Tooling Considerations

Because C++23 is still relatively new, **compiler support may vary** across GCC, Clang, and MSVC. Readers are advised to:

- Check the compiler's feature test macros (`__cpp_...`) when using advanced functionality.
- Keep compilers updated to their latest stable releases for maximum support.
- Be aware of build systems (CMake, Meson) that may require explicit standard version flags (e.g., `-std=c++23`).

This book highlights cases where certain features may not yet be universally available, ensuring that readers are prepared for practical limitations.

How Educators Can Use This Book

This book is also structured to be valuable in **academic or training contexts**:

- **Course Material**: Each section can serve as a lecture module, complete with examples.
- **Assignments**: Many examples can be extended into exercises for students.
- **Reference**: Appendices at the end provide quick-access tables and summaries suitable for exams or coding interviews.

Summary

The goal of this guide is not only to teach the **new features of C++23** but also to help readers **integrate them confidently into their programming practices**.

Whether you choose to read it cover-to-cover as a student of modern C++, or consult it selectively as a professional reference, this book is designed to provide both **clarity and depth**.
Approach the material actively: experiment with examples, compare old and new approaches, and apply the techniques in real projects. Doing so will ensure that the knowledge gained here becomes part of your professional C++ skillset.

Audience

Every book has an intended readership, and this one is no exception. *Mastering Modern C++23 – A Complete Guide to the Latest Standard* has been written with a **specific audience in mind**: developers who already know C++ and wish to upgrade their skills to align with the modern features introduced in the C++23 standard.

Who This Book Is For

This book is primarily aimed at:

- **Intermediate C++ Programmers**
  Those who are familiar with the basics of C++ — such as variables, functions, classes, inheritance, templates, and memory management — but may not yet have a strong grasp of the advanced features introduced in recent standards. These readers will benefit from clear explanations of modern syntax, standard library utilities, and migration strategies that connect traditional practices with modern techniques.

- **Advanced C++ Developers**
  Experienced programmers who are already comfortable with C++11, C++14, or C++17 (and possibly some C++20) but want to **deepen their understanding**

of C++23. For them, the book provides not only the technical details of each new feature but also insights into **design rationale, performance considerations, and best practices**.

- **System Programmers and Performance-Critical Developers**
  Those working on **embedded systems, operating systems, compilers, high-performance computing, and financial platforms** will find the coverage of compile-time programming, memory management updates, and low-level utilities especially relevant.

- **Library and Framework Designers**
  Programmers who create reusable components, libraries, or frameworks will gain practical knowledge on advanced features such as expanded `constexpr`, metaprogramming utilities, ranges, and flat containers that directly impact the design of robust APIs.

- **Educators and Trainers**
  Teachers, trainers, and mentors looking for structured material to help others transition from older versions of C++ to modern C++23 will find this book organized in a way that can be adapted for classroom, workshop, or self-study use.

Who This Book Is Not For

This book is **not designed for absolute beginners**. Readers who have no prior experience in C++ or programming will find the material too advanced. While the book explains new features in detail, it assumes knowledge of:

- Core C++ syntax (control structures, classes, functions, pointers, references).
- Object-oriented programming principles.

- Templates and generic programming at a basic level.
- Familiarity with C++11 or later (smart pointers, move semantics, basic lambdas).

Those seeking an introduction to C++ from scratch should begin with a general introductory textbook before approaching this volume.

What Readers Can Expect to Gain

By working through this book, readers will:

- **Understand Every Major Change in C++23**
  From new language constructs (such as deducing `this` and multidimensional subscripts) to library innovations (`std::expected`, `std::print`, ranges adaptors, `mdspan`, and stacktrace support).
- **Learn Best Practices**
  Each chapter not only explains features but also provides **guidelines, idioms, and cautions** to help avoid common pitfalls and write code that is both modern and maintainable.
- **Gain Migration Strategies**
  For developers maintaining legacy codebases, the book highlights how to adopt C++23 features incrementally while preserving compatibility with existing systems.
- **Develop a Future-Ready Skillset**
  C++23 is a bridge between the revolutionary changes of C++20 and the ongoing evolution toward C++26 and beyond. Mastering it ensures that readers are prepared for future updates to the standard.

Why This Audience Matters

C++ continues to be the language of choice in domains where **performance, efficiency, and reliability** are paramount. The global demand for developers who can **write modern, safe, and maintainable C++ code** is increasing, not decreasing. By targeting intermediate and advanced programmers, this book contributes to a professional community that is expected to:

- Maintain and modernize critical software infrastructure.
- Lead projects that adopt newer standards of C++.
- Educate the next generation of C++ developers.

Summary

This book has been crafted for developers who are already fluent in C++ fundamentals and now want to **upgrade their knowledge to C++23**. Whether you are working on system-level software, high-performance computing, financial systems, or designing modern libraries, this book will serve as both a **guide to learning** and a **reference for practice**.
The audience is not defined by beginners but by those who already know C++ and are ready to master the **modern standard**.

Part I

Foundations of C++23

Chapter 1

Evolution of the C++ Language

1.1 From C++98 → C++11 → C++14 → C++17 → C++20 → C++23

C++ has always been a language of balance: balancing performance with abstraction, safety with control, and stability with innovation. Its standardization process reflects this balance, evolving carefully over decades to meet the changing needs of software development. To fully understand the significance of C++23, it is essential to trace the journey of the language through its previous major milestones.

1.1.1 The First International Standard: C++98

C++ was standardized for the first time in **1998** with the publication of ISO/IEC 14882:1998, commonly referred to as **C++98**.
Before this milestone, C++ had already become popular through compilers like Borland C++ and Microsoft Visual C++, but the lack of a unified standard led to inconsistencies across platforms.

Key features of **C++98** included:

- **Templates**: Formalized as a cornerstone of generic programming.
- **Exceptions**: Introduced structured error handling through `try`, `catch`, and `throw`.
- **Namespaces**: Added to reduce name collisions in large projects.
- **Standard Template Library (STL)**: Vector, list, map, algorithms, and iterators became part of the official standard library.

C++98 provided a stable foundation but was conservative. It focused primarily on **standardization of existing practices** rather than innovation.

1.1.2 C++03: A Minor Revision

The **C++03** standard (ISO/IEC 14882:2003) was not revolutionary. It served mainly as a bug-fix and clarification release to address issues from C++98.
Notable adjustments included improvements to template rules and clearer specifications for library behavior.
While not significant in terms of new features, C++03 ensured that C++ had a **more precise and reliable standard**, preparing the way for the major innovations that followed.

1.1.3 C++11: The "Modern C++" Revolution

The **2011 standard**, often called **C++11**, represented a **turning point** in the language's history. Sometimes referred to as "C++0x" during its long development, it transformed C++ into a language capable of competing with modern programming paradigms.

Major features of **C++11** included:

- **Move Semantics and Rvalue References** (`T&&`): Revolutionized performance optimization by enabling efficient resource transfers.
- **Auto Type Deduction**: Simplified variable declarations (`auto i = 5;`).
- **Range-based For Loops**: Cleaner iteration syntax.
- **Lambda Expressions**: Allowed inline function objects with closures.
- **Smart Pointers (`std::unique_ptr`, `std::shared_ptr`)**: Safer memory management.
- **Concurrency Support**: Threads, mutexes, futures, and condition variables.
- **Uniform Initialization**: Braced initializer lists for consistency.
- **constexpr Functions**: Computation at compile time.
- **nullptr**: A type-safe null pointer constant.
- **Strongly-Typed Enums (`enum class`)**: Scoped and type-safe enumerations.

C++11 was considered the **true modernization** of C++. It marked the beginning of what is now referred to as **Modern C++**.

1.1.4 C++14: Refinement and Simplification

The **2014 standard**, **C++14**, was deliberately smaller in scope. It aimed to refine C++11 by fixing issues, simplifying usability, and making the new features more consistent.
Key improvements in **C++14** included:

- **Generic Lambdas**: Allowed lambdas with auto parameters.
- **Return Type Deduction for Functions**: Made code less verbose.
- **std::make_unique**: Provided a safer way to create unique pointers.
- **Relaxed constexpr Restrictions**: Allowed more complex computations at compile time.

C++14 was not a revolutionary release but served as a **stabilization step**, ensuring that the bold innovations of C++11 could be reliably used in production.

1.1.5 C++17: Consolidation and New Tools

The **2017 standard**, **C++17**, provided another wave of usability enhancements and standard library expansions. While less dramatic than C++11, it consolidated modern practices and introduced powerful new capabilities.
Key features of **C++17** included:

- **Structured Bindings**: Simplified tuple and pair unpacking (`auto [x, y] = foo();`).
- **if and switch with Initializers**: Cleaner conditional control (`if (auto it = map.find(k); it != map.end())`).
- **std::optional, std::variant, std::any**: New vocabulary types.
- **std::string_view**: Non-owning string references for performance.
- **Parallel Algorithms**: Standard algorithms with parallel execution policies.
- **Inline Variables**: Simplified handling of header-defined constants.

- **Filesystem Library**: Standardized file and directory operations.

C++17 made the language more **practical and expressive**, continuing the push toward safety and productivity without sacrificing performance.

1.1.6 C++20: The Second Great Leap

The **2020 standard**, **C++20**, is often considered the **largest and most impactful revision since C++11**. It introduced features that dramatically changed the way C++ code could be written, often compared to the effect of generics in Java or LINQ in C#.
Major features of **C++20** included:

- **Concepts**: A formal way to constrain templates, making generic programming clearer and safer.
- **Ranges Library**: Unified way to handle sequences with pipelines (`views::filter`, `views::transform`).
- **Coroutines**: Foundation for asynchronous programming and generators.
- **Modules**: A new way to organize and import code, reducing compile times.
- **constexpr Expansion**: Many standard library functions became usable at compile time.
- **Three-Way Comparison Operator (`<=>`)**: Simplified comparison logic.
- **Calendar and Time Zones Library**: Extended `<chrono>` capabilities.
- **Improved Lambdas**: Lambdas with template parameters.

C++20 was a **transformational release**, redefining how developers approached code organization, generic programming, and asynchronous operations.

1.1.7 C++23: Evolution with Practical Refinement

The **2023 standard**, **C++23**, builds on the foundations of C++20 but is more evolutionary than revolutionary. Its focus is on **refinement, consistency, and usability**.
Highlights of **C++23** include:

- **Language Features**: Deduction of `this`, multidimensional subscripts, static lambdas, new preprocessor directives, expanded `constexpr` rules, and portable UTF-8 source encoding.
- **Standard Library Additions**:
  - `std::expected` for safer error handling.
  - `std::print` and `std::println` for modern output.
  - New containers like `std::flat_map`, `std::flat_set`, and `std::mdspan`.
  - Expanded range adaptors such as `views::zip`, `views::enumerate`, and `views::cartesian_product`.
  - `std::generator` for coroutine-based iteration.
  - Stacktrace support for diagnostics.
- **Refinements**: Lifetime extensions, constexpr improvements, additional string utilities, and memory management enhancements (`std::out_ptr`, `std::allocate_at_least`).
- **Obsolete Features Removed**: Elimination of garbage collection support and some problematic wide character literal behaviors.

C++23 is therefore less about introducing radical new paradigms and more about **strengthening the language, expanding its usability, and polishing rough edges left by C++20**.

1.1.8 The Trajectory of C++

Looking across these standards, a pattern emerges:

- **C++98 and C++03**: Foundation and stability.
- **C++11**: Modernization and transformation.
- **C++14 and C++17**: Refinement and expansion.
- **C++20**: Revolutionary leap in capability.
- **C++23**: Practical evolution and usability improvements.

This trajectory reflects the philosophy of C++: to **innovate carefully, evolve consistently, and preserve performance and compatibility**. Each standard builds on the last, ensuring that developers who master the current standard are prepared for the future.

1.2 Philosophy: Safety, Efficiency, and Expressiveness

From its inception, C++ has been driven by a philosophy that distinguishes it from other programming languages: a deep respect for **performance, control, and abstraction without compromise**. Every revision of the C++ standard has been guided by this philosophy, even as new demands in hardware, software, and industry practices shaped the language's growth.
In modern C++, and particularly with the arrival of C++23, three principles stand out as central pillars: **Safety, Efficiency, and Expressiveness**.

1.2.1 Safety

C++ has always provided developers with **direct access to hardware and memory**. This control is powerful but historically came with risks: memory leaks, dangling pointers, buffer overflows, and undefined behavior. Over time, the standard has introduced tools to **mitigate these risks while preserving control**.

- **Memory Safety**
  - Early C++ required manual memory management with `new` and `delete`.
  - C++11 introduced **smart pointers** (`std::unique_ptr`, `std::shared_ptr`) to manage lifetimes automatically.
  - Later standards expanded safety with `std::weak_ptr`, stricter type rules, and `std::make_unique`.
  - C++23 adds utilities such as `std::out_ptr` and `std::inout_ptr` to safely bridge C++ smart pointers with C APIs.
- **Type Safety**

  - Features like `enum class` (C++11), `std::variant` (C++17), and concepts (C++20) reduce the possibility of type misuse.
  - C++23 continues this effort by introducing clearer conversions (e.g., narrowing conversions in `static_assert` and `if constexpr`) and improving diagnostics with **stacktrace support**.

- **Error Safety**
  - Traditional error handling with return codes was error-prone.
  - `throw`/`catch` improved safety but introduced runtime overhead.
  - Modern C++ embraces **monadic types** like `std::optional` (C++17) and `std::expected` (C++23), allowing errors to be represented in type-safe, explicit ways.

Safety in C++ does not mean removing responsibility from the programmer; instead, it means providing **tools that help avoid mistakes without taking away control**.

1.2.2 Efficiency

Efficiency is the **non-negotiable principle** of C++. From embedded microcontrollers to high-frequency trading systems, C++ remains the language of choice when performance matters.

- **Zero-Cost Abstractions**
  A guiding principle of C++ is that higher-level features must not impose hidden runtime costs if they are not used. Templates, inline functions, and move semantics are designed with this in mind.

- **Compile-Time Computation**

  - C++11 introduced `constexpr`, allowing limited evaluation at compile time.
  - Each subsequent standard expanded this, culminating in C++20 and C++23 where **entire algorithms and utilities can be computed at compile time**. This reduces runtime overhead while retaining flexibility.

- **Parallelism and Concurrency**
  - C++11 introduced the thread library.
  - C++17 added parallel algorithms.
  - C++20 and C++23 build on these foundations with ranges, coroutines, and generators, enabling expressive yet efficient concurrent solutions.

- **Memory and Data Access**
  - Efficiency is not only about CPU cycles but also about memory layout and cache usage.
  - C++23 introduces `std::mdspan` (a non-owning multidimensional array view) and flat containers (`std::flat_map`, `std::flat_set`), both optimized for performance in modern workloads.

In C++, efficiency means **getting as close as possible to the hardware** while allowing the compiler to optimize aggressively without unnecessary abstraction penalties.

1.2.3 Expressiveness

While C++ has a reputation for complexity, its evolution shows a consistent effort to make the language **more expressive, readable, and maintainable** without losing its power.

- **Clarity of Intent**
  Features such as range-based for loops (C++11), structured bindings (C++17), and the `<=>` three-way comparison operator (C++20) reduce boilerplate code and make programs easier to reason about.

- **Powerful Abstractions**

  - Templates were once considered arcane; with concepts (C++20), they are now easier to constrain and reason about.
  - C++23 introduces **deducing `this`**, making member functions more expressive when dealing with fluent interfaces and functional-style APIs.

- **Modern Standard Library Additions**

  - `std::print` (C++23) makes formatted output simpler and safer than legacy `printf` or verbose `std::cout` streams.
  - Expanded ranges adaptors (`views::zip`, `views::enumerate`, `views::cartesian_product`) allow concise and expressive iteration patterns.

- **Readable and Maintainable Code**
  Expressiveness in modern C++ is not about writing fewer characters but about writing **clearer, safer, and more maintainable code**. By expanding the library and improving syntax, C++23 makes complex patterns approachable without sacrificing precision.

1.2.4 Balancing the Triad

The enduring philosophy of C++ can be summarized as a **balance of safety, efficiency, and expressiveness**:

- Safety ensures programs behave correctly and predictably.
- Efficiency ensures they run fast and use resources wisely.
- Expressiveness ensures developers can communicate intent clearly and build maintainable systems.

Other languages often emphasize one principle at the cost of others. C++ stands apart in its commitment to **all three simultaneously**. This is why C++ remains central to industries where correctness and performance cannot be compromised — from finance to game engines, from real-time systems to scientific computing.

1.2.5 C++23 in Context

C++23 does not reinvent the language but continues its **philosophical journey**:

- It strengthens **safety** with better diagnostics, stricter rules, and error-handling types.
- It advances **efficiency** with new containers, memory management functions, and constexpr expansions.
- It enhances **expressiveness** with deducing `this`, modern I/O utilities, and expanded ranges.

In this way, C++23 represents the natural continuation of the vision Bjarne Stroustrup set out decades ago: a language that empowers programmers to write code that is **fast, safe, and elegant — all at once**.

1.3 Why Upgrade to C++23 Now?

The release of a new C++ standard is always a milestone, but C++23 represents a particularly **compelling reason for developers to upgrade now**. While previous standards laid the groundwork for modern C++ programming, C++23 consolidates these innovations, adds practical improvements, and closes gaps that previously limited efficiency, safety, and expressiveness.

Understanding **why an upgrade is timely** requires examining both **developer needs** and **industry trends**.

1.3.1 Benefit from Modern Language Features

C++23 introduces features that simplify coding patterns and reduce boilerplate:

- **Deduction of `this`** in member functions improves code readability and allows fluent APIs.
- **Multidimensional subscript operators** simplify complex data structure access.
- **Static lambdas and operator() support** streamline compile-time computations and metaprogramming.
- **Improved `constexpr` support** allows more computations at compile time, reducing runtime overhead.

By upgrading, developers gain immediate access to these **language-level improvements**, allowing code that is **simpler, faster, and less error-prone**.

1.3.2 Expanded and Safer Standard Library

C++23 expands the standard library in ways that directly improve safety and efficiency:

- **`std::expected`** provides a type-safe, standardized way to return errors without exceptions.
- **Flat containers (`std::flat_map`, `std::flat_set`)** improve cache efficiency and speed.
- **`std::mdspan`** enables multidimensional array handling with zero overhead.
- **Enhanced ranges adaptors** like `views::zip`, `views::enumerate`, and `views::cartesian_product` make iteration concise and expressive.
- **Formatted printing with `std::print`** and **span-based I/O (`std::spanstream`)** simplify I/O operations while reducing common errors.

These library improvements allow developers to write modern C++ code **without relying on external dependencies or complex workarounds**, increasing maintainability and long-term reliability.

1.3.3 Improved Safety and Reliability

C++23 emphasizes safety in ways that directly impact real-world development:

- **Better diagnostics**: Stacktrace utilities provide actionable information during exceptions or program crashes.
- **Lifetime extensions for temporaries**: Avoid subtle bugs in range-based loops.

- **Explicit lifetime management** via `std::start_lifetime_as` reduces undefined behavior in low-level code.

- **Standardized error handling** with `std::expected` encourages explicit, safe error propagation.

By upgrading, developers **reduce the risk of runtime errors, undefined behavior, and memory-related issues**, which is particularly important for large, long-lived codebases.

1.3.4 Performance Enhancements

C++23 builds on C++20's emphasis on efficiency, offering tools that optimize both CPU and memory usage:

- **Compile-time computations**: Expanded `constexpr` capabilities allow more logic to be evaluated at compile time.

- **Flat containers and `mdspan`**: Optimize memory access patterns for better cache utilization.

- **Move-only function wrappers (`std::move_only_function`)** reduce overhead for high-performance applications.

- **Parallelism-ready ranges**: Seamless integration with parallel algorithms reduces development complexity in multi-threaded programs.

Upgrading now allows developers to **leverage these optimizations** without waiting for C++26, ensuring applications are **fast, lean, and future-proof**.

1.3.5 Better Alignment with Industry Standards

Adopting C++23 ensures teams are aligned with:

- **Modern tooling**: Compilers, static analyzers, and IDEs increasingly optimize for C++20 and C++23 features.
- **Cross-platform consistency**: C++23 ensures code behaves predictably on Windows, Linux, and embedded platforms.
- **Future maintainability**: Using modern standards reduces technical debt and simplifies onboarding for new developers.

In industries such as finance, automotive, aerospace, and high-performance computing, **upgrading to C++23 is not just a convenience but a strategic necessity**.

1.3.6 Reduced Technical Debt

Legacy C++ code often contains patterns that are now **deprecated or unsafe**. By upgrading:

- Developers can **replace manual memory management** with smart pointers and safer constructs.
- Old loops, manual tuple unpacking, and verbose error handling can be replaced with **modern, maintainable patterns**.
- Deprecated features removed in C++23 (such as certain wide-character literal behaviors) **prevent hidden bugs** from persisting.

Upgrading proactively reduces long-term maintenance costs and ensures **software longevity**.

1.3.7 Conclusion

C++23 represents a **mature, refined, and practical evolution** of the C++ language. Developers who adopt it now benefit from:

- Simplified and safer coding patterns.
- Expanded standard library tools for error handling, iteration, and I/O.
- Optimized performance for modern hardware.
- Alignment with industry best practices.

Upgrading is not just about using new syntax; it is about **adopting a more reliable, expressive, and efficient programming model**. By embracing C++23 today, developers position themselves and their codebases for **long-term maintainability, performance, and future-proof design**.

Chapter 2

What's New in C++23 (High-Level Overview)

2.1 Language Features

C++23 is the latest iteration of the C++ standard, representing a **mature, evolutionary step** rather than a radical departure. Its primary goal is to refine, extend, and make modern C++ **safer, more expressive, and easier to use** while maintaining the unparalleled performance that C++ is known for.
This chapter introduces the **key language features** of C++23, providing a high-level overview before diving into deeper technical details in later chapters.

2.1.1 Deducing `this`

One of the most anticipated language features in C++23 is **deducing `this`**, also referred to as *explicit object parameters*. This feature allows member functions to **explicitly declare the type of the object they operate on**, enhancing clarity and

flexibility.
Benefits include:

- Easier creation of **fluent APIs** and chainable function calls.
- Improved **const-correctness** and reference qualification for member functions.
- Support for **generic member functions** without verbose template syntax.

Example:

```
struct S {
    void print(this S& self) { std::cout << self.value << "\n"; }
    int value;
};
```

This innovation increases **expressiveness** and reduces boilerplate in modern C++ codebases.

2.1.2 Multidimensional Subscript Operators

C++23 allows **overloading of multidimensional subscript operators**, enabling more intuitive access to complex data structures such as matrices or multi-dimensional arrays.
Example usage:

```
Matrix m;
m[1, 2, 3] = 42;
```

This makes code involving **multi-indexed structures** significantly cleaner and more readable.

2.1.3 Static Lambdas and Static Operator Overloads

C++23 introduces **static lambdas** and allows `static operator[]` and `static operator()` definitions. These enhancements:

- Enable **compile-time callable objects** without capturing external state.
- Improve **metaprogramming capabilities**.
- Reduce runtime overhead when objects do not require state capture.

2.1.4 `auto(x)` and `auto{x}` – Decay-Copy in the Language

C++23 expands `auto` deduction with **decay-copy semantics**, simplifying initialization:

- Ensures that temporary objects are **safely copied or moved** into variables.
- Reduces subtle lifetime and type issues that often appear with templates and forwarding.

2.1.5 Assumptions via `[[assume(expression)]]`

The new **`[[assume]]` attribute** allows the programmer to inform the compiler of invariants or assumptions. Benefits include:

- **Enabling optimizations** when certain conditions are guaranteed.
- Improving **static analysis and correctness checks**.

Example:

```
[[assume(x > 0)]]
void process(int x) { /* compiler can optimize based on assumption */ }
```

2.1.6 Attributes on Lambda Expressions

C++23 allows **attaching attributes to lambda expressions**, enhancing readability and correctness:

- Enables `[[nodiscard]]`, `[[likely]]`, and other standard attributes directly on lambdas.
- Simplifies **function annotations** in functional-style programming.

2.1.7 Extended Floating-Point Types

C++23 introduces **optional extended floating-point types**:

- `std::float16_t`, `std::float32_t`, `std::float64_t`, `std::float128_t`
- `std::bfloat16_t`

These types provide:

- **Improved precision** where needed.
- **Hardware-specific optimizations**, particularly in scientific computing and machine learning.

2.1.8 New Preprocessor Directives

C++23 adds **modern preprocessor capabilities**:

- `#elifdef` and `#elifndef` for cleaner conditional compilation.
- `#warning` to emit compiler warnings during preprocessing.
- Improved whitespace handling before line splicing.

These changes **simplify conditional compilation** and improve code maintainability.

2.1.9 Literal Suffixes for `std::size_t`

A new **`uz` suffix** allows `std::size_t` literals:

```
auto n = 42uz;
```

This small addition improves **clarity and type safety** when working with container sizes and indexing.

2.1.10 Simplified Move Semantics

C++23 introduces **simpler implicit move semantics**, reducing boilerplate when returning objects by value and enabling **more efficient code generation** without explicit move statements.

2.1.11 Lifetime Management Enhancements

- **Extended lifetime of temporaries** in range-based for loop initializers.
- **`std::start_lifetime_as`** allows explicit control over the lifetime of implicit-lifetime types.

These changes **prevent subtle bugs** and make object lifetimes safer in complex expressions.

2.1.12 CTAD from Inherited Constructors

Class Template Argument Deduction (CTAD) is now supported with **inherited constructors**, making generic classes and template inheritance **more intuitive and less verbose**.

2.1.13 Lambda Improvements

- **Optional parentheses** for lambdas with no parameters.
- More **flexible attributes** and constexpr support.

These refinements make functional programming patterns in C++ **more expressive and concise**.

2.1.14 Constexpr Enhancements

C++23 further expands **constexpr capabilities**:

- Non-literal variables, labels, and even `goto` are allowed in constexpr functions.
- Static and `thread_local` variables can be evaluated at compile time.
- Functions no longer require literal-type return values, increasing **compile-time flexibility**.

2.1.15 Text Encoding and Universal Character Escapes

C++23 improves **source code portability and readability**:

- UTF-8 support as portable source encoding.
- Named universal character escapes, e.g., `"\N{CAT FACE}"`.
- Delimited escape sequences: `\o{7777}`, `\x{CODE}`, `\u{CAFE}`.

This ensures **modern internationalization support** in applications.

2.1.16 `if consteval` / `if not consteval`

C++23 introduces **compile-time conditional expressions**:

- Distinguish code paths executed at compile time versus runtime.
- Enhance template and constexpr programming by providing **clear, concise branching**.

2.1.17 Summary

C++23 introduces a mix of **language refinements, safety improvements, performance optimizations, and expressiveness enhancements**. Unlike prior revolutionary updates, C++23 focuses on **practical usability**, addressing **real-world coding challenges** while preserving the language's core philosophy: **performance, control, and expressiveness**.
This high-level overview sets the stage for subsequent chapters, where each feature will be explored in depth, with **detailed syntax, use cases, and best practices** for professional C++23 development.

2.2 Library Features

The C++ standard library has always been a **core strength of the language**, providing developers with **ready-to-use, efficient, and safe abstractions**.
With C++23, the standard library receives a **significant set of new features, enhancements, and containers**, enabling developers to write cleaner, faster, and safer code.
This section outlines the **most impactful library features**, grouped by functionality.

Core Utilities and Language Support

`std::expected`

- Introduced in `<expected>`, `std::expected<T, E>` provides a **type-safe alternative to exceptions** for error handling.
- Represents either a **value of type T** or an **error of type E**, making it explicit when functions can fail.
- Supports **monadic operations** (`transform`, `or_else`, `and_then`) for chaining computations safely.

Example:

```
std::expected<int, std::string> divide(int a, int b) {
    if(b == 0) return std::unexpected("division by zero");
    return a / b;
}
```

`std::move_only_function`

- A **move-only callable wrapper** that replaces `std::function` when copy semantics are not required.
- Reduces runtime overhead for **high-performance callbacks**.

`std::bind_back`

- A **utility for binding arguments** to the end of a callable.
- Complements `std::bind_front` (C++20), enhancing **functional programming flexibility**.

`std::byteswap`

- A utility function to **reverse the byte order** of integral types.
- Essential for **endianness conversions** and low-level systems programming.

`std::invoke_r` and `std::forward_like`

- `std::invoke_r` allows **invoking a callable and forcing a return type**.
- `std::forward_like` improves **generic forwarding** in templates for consistency and safety.

`std::to_underlying`

- Retrieves the **underlying integer value** of an enum.
- Eliminates unsafe manual casts, improving **type safety**.

2.2.1 Coroutines and Generators

`std::generator`

- Introduced as a **synchronous coroutine generator** in `<generator>`.
- Provides **range-based iteration** over values produced lazily, similar to Python generators.
- Supports modern **pipeline and functional programming patterns**.

Example:

```
std::generator<int> range(int start, int end) {
    for(int i = start; i < end; ++i) co_yield i;
}
```

2.2.2 Diagnostics

Stacktrace Library

- `<stacktrace>` provides **portable access to call stacks** for debugging and error reporting.
- Useful for **runtime diagnostics** and improving maintainability in large applications.

2.2.3 Ranges and Algorithms

C++23 expands the ranges library introduced in C++20 with **new adaptors and algorithms**:

- **New Range Adaptors**
  - `views::adjacent` and `views::adjacent_transform`
  - `views::as_const`, `views::as_rvalue`
  - `views::cartesian_product`, `views::chunk`, `views::chunk_by`
  - `views::enumerate`, `views::join_with`, `views::repeat`, `views::slide`, `views::stride`
  - `views::zip` and `views::zip_transform`
- **Range Conversion and Utilities**
  - `ranges::to` for converting ranges to containers
  - Relaxed range adaptors for **move-only types**
- **New Constrained Algorithms**
  - `ranges::starts_with`, `ranges::ends_with`
  - `ranges::contains`, `ranges::contains_subrange`
  - `ranges::find_last`, `ranges::find_last_if`, `ranges::find_last_if_not`
  - `ranges::iota`, `ranges::shift_left`, `ranges::shift_right`, `ranges::fold_left`

These enhancements make **complex data manipulations** more concise, readable, and efficient.

2.2.4 Containers

`std::mdspan`

- A **non-owning multidimensional array reference** introduced in `<mdspan>`.
- Enables **efficient views on arrays** without copying data.
- Supports arbitrary dimensions, layouts, and strides.

Flat Containers

- `std::flat_map`, `std::flat_multimap`, `std::flat_set`, `std::flat_multiset`
- Containers **optimized for memory locality**, improving **cache performance**.
- Provide **sorted associative functionality** over underlying random-access containers.

Stack and Queue Improvements

- Iterator-pair construction is now supported, simplifying **initialization from ranges**.

Heterogeneous Erasure Overloads

- Associative containers can now handle **heterogeneous keys** efficiently, improving usability and type safety.

2.2.5 Memory Management

Smart Pointer Adaptors

- `std::out_ptr` and `std::inout_ptr` bridge **C++ smart pointers with C APIs** safely.

Allocator Enhancements

- `std::allocate_at_least` and `std::allocator::allocate_at_least` provide **guaranteed allocation sizes**.
- `std::start_lifetime_as` allows **explicit lifetime management** for implicit-lifetime types.

Restriction on `std::allocator_traits`

- User specialization is now **disallowed**, improving **standard compliance and safety**.

2.2.6 String and Text Processing

- **New member functions**: `std::basic_string::contains`, `std::basic_string_view::contains`
- **Explicit range constructors** for `basic_string_view`
- **`resize_and_overwrite`** for efficient string modification
- **Rvalue reference overloads** for `substr`
- **Formatting support**: Ranges, tuples, escaped characters, `std::thread::id`, and stacktraces

2.2.7 I/O and Print

- **`std::print` and `std::println`** simplify formatted output, replacing verbose streams or unsafe printf calls.
- **`spanstream`**: Span-based string streams for **memory-efficient I/O**.

- **Volatile pointer printing** and exclusive mode support in `fstream` improve low-level file operations.

2.2.8 Compile-Time Support

- Expanded `constexpr` support for **bitsets, unique_ptr, type_info, and `<cmath>` functions**
- Integral overloads of `std::to_chars` and `std::from_chars` enable **compile-time numeric conversions**

2.2.9 Summary

C++23's library enhancements represent a **significant leap in usability, safety, and performance**:

- **Error handling and diagnostics** are now more robust (`std::expected`, `stacktrace`).
- **Data manipulation** is more expressive (`ranges`, `mdspan`, `flat containers`).
- **I/O and formatting** are simpler and safer (`std::print`, `spanstream`).
- **Memory and lifetime management** tools reduce subtle bugs and improve reliability.

By combining these features with the new language features, C++23 provides a **comprehensive, modern toolkit** for building safe, efficient, and maintainable software.

2.3 Removed and Deprecated Features

While C++23 introduces numerous new capabilities, it also **removes outdated features and deprecates others** to **enhance safety, consistency, and clarity**. Removing or deprecating certain language constructs ensures that developers **adopt modern, safer, and maintainable programming patterns**.

2.3.1 Removed Features

Garbage Collection Support and Reachability-Based Leak Detection

- C++23 **removes optional garbage collection support**, including reachability-based leak detection.
- Rationale: The feature was rarely used in production code and added **complexity and potential runtime overhead**.
- Developers should continue using **smart pointers** (`std::unique_ptr`, `std::shared_ptr`) and RAII for memory management.

Mixed Wide String Literals Concatenation

- Previously, concatenating wide and narrow string literals like `const auto* s = u"q" U"p";` was allowed.
- C++23 makes this **ill-formed**, enforcing consistent **string literal encoding**.
- Ensures **predictable behavior** across compilers and platforms.

Non-Encodable and Multicharacter Wide Character Literals

- Literals such as `wchar_t x = 'db';` are now **ill-formed**.
- This removes ambiguity in **wide character representation** and improves **cross-platform safety**.

2.3.2 Deprecated Features

`std::aligned_storage` and `std::aligned_union`

- Deprecated because modern alternatives exist:
  - `alignas` specifier for alignment.
  - `std::byte` and `std::array` for raw storage.
- Encourages **type-safe and standard-compliant memory management**.

`std::numeric_limits::has_denorm`

- Deprecated in favor of **more robust and platform-consistent floating-point diagnostics**.

2.3.3 Reverted Deprecations

Comma Operator in Subscript Expressions

- Previously deprecated for multidimensional subscript overloading.
- C++23 **reintroduces the operator**, but with **revised semantics**:
  - Enables **overloadable multidimensional subscript operators** like `v[1, 2, 3]`.
- This balances **backward compatibility** with **modern expressive patterns**.

Some C Headers for C Compatibility

- Certain `<*.h>` headers remain for **legacy C compatibility**, but their use is **discouraged in modern C++ code**.
- Developers are encouraged to use the **modern `<c\*>` headers** instead.

2.3.4 Rationale Behind Removals and Deprecations

The decisions to **remove or deprecate features** in C++23 reflect three main principles:

1. **Safety**
    - Deprecated or removed features often lead to **undefined behavior**, subtle bugs, or cross-platform inconsistencies.
2. **Simplicity and Clarity**
    - Removing outdated constructs reduces **language complexity**, making C++ easier to learn and maintain.
3. **Encouragement of Modern Practices**
    - Developers are guided toward **safe, expressive, and maintainable alternatives** such as:
        - Smart pointers over manual memory management
        - `alignas` over `aligned_storage`
        - Standardized string literals and encoding practices

2.3.5 Migration Guidelines

To modernize codebases in light of these changes:

- **Replace deprecated memory utilities**:

```
// Old
std::aligned_storage<sizeof(T), alignof(T)>::type buffer;
// Modern
alignas(T) char buffer[sizeof(T)];
```

- **Avoid mixing string literal types**: Ensure all concatenated strings share the same encoding.
- **Move from wide character multicharacter literals**: Use `char32_t` or Unicode escapes instead.
- **Replace `std::numeric_limits::has_denorm` checks**: Use **`std::fpclassify`** and modern floating-point diagnostics.
- **Adopt `std::expected` for error handling** instead of older exception or status-code patterns where possible.

2.3.6 Conclusion

C++23's removal and deprecation strategy is **forward-looking**, aiming to:

- Reduce **legacy baggage** that complicates learning and maintenance.
- Increase **runtime safety and predictability**.
- Encourage **modern, maintainable C++ coding practices**.

Understanding what has been removed or deprecated is critical for:

- Migrating existing codebases smoothly.
- Writing future-proof C++23 code.
- Leveraging the **full power of the new standard** safely and efficiently.

2.4 Tooling and Compiler Support

C++23 represents the **latest stable iteration of the C++ standard**, but its practical utility depends on **compiler implementations, IDE support, and build toolchains**. This chapter highlights the current status of **C++23 adoption in compilers, static analyzers, and other development tools**, along with guidance for developers upgrading from earlier standards.

2.4.1 Compiler Support

GCC (GNU Compiler Collection)

- GCC has **partial support for C++23**, with many core features implemented in versions **12 and later**, including:
  - `std::expected` and `std::mdspan`
  - Expanded `constexpr` support
  - Ranges and new algorithms
  - Language features like **deducing `this`** and **static lambdas**
- **GCC 13+** offers more comprehensive support for the majority of C++23 features.

Clang / LLVM

- Clang 16 and above implements most C++23 features:
  - **Core language enhancements**: deducing `this`, multidimensional subscript operator

  - **Library support**: `std::generator`, `stacktrace`, `flat_map/set`, `ranges` improvements
  - **Attributes** and **constexpr improvements**

- Clang emphasizes **compile-time diagnostics and static analysis**, making it ideal for modern C++23 adoption.

MSVC (Microsoft Visual C++)

- MSVC fully supports most C++23 features starting with **Visual Studio 2022 v17.4+**.
- Includes:
  - Core language enhancements (e.g., CTAD from inherited constructors)
  - Library updates (e.g., `std::expected`, ranges adaptors, `mdspan`)
  - New diagnostics and formatting support (`std::print`)

Intel oneAPI / ICC

- Intel compilers have **experimental support** for C++23 features, focusing on **performance-critical code**.
- Recommended for **high-performance computing and numerical applications**.

Other Compilers

- Smaller or niche compilers (e.g., **Embarcadero C++Builder, ARM compilers**) are gradually integrating C++23 features.
- Developers targeting embedded systems should verify **which features are supported and optimize usage accordingly**.

2.4.2 Build Tools and Standard Library Implementation

CMake

- CMake 3.26+ officially supports **C++23 as a language standard**, allowing developers to specify:

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

- Facilitates **cross-platform builds** and **compiler-specific feature checks**.

Package Managers and Standard Library Versions

- **vcpkg**, **Conan**, and **CPM.cmake** now provide **C++23-compliant libraries**.
- Standard library updates include:
  - **libc++** (Clang) C++23 headers
  - **libstdc++** (GCC) updates with `std::expected`, `ranges`, `mdspan`
  - MSVC STL updates included in latest Visual Studio 2022 updates

2.4.3 IDE Support

Visual Studio

- Visual Studio 2022 fully supports **C++23 syntax highlighting, code completion, and refactoring tools**.
- Integrated **static analysis** supports the new standard's safety and constexpr features.

CLion / JetBrains IDEs

- CLion 2023+ provides **C++23 project templates** and **compiler feature detection** for GCC and Clang.

VS Code

- With **C++ extensions**, VS Code supports C++23 features through **compiler integration** (GCC, Clang, MSVC).
- Provides **linting, code formatting, and IntelliSense for modern C++ constructs**.

2.4.4 Static Analysis and Modern Tooling

- **clang-tidy**: Updated with C++23 checks, including deprecated features, lifetime rules, and constexpr diagnostics.
- **cppcheck**: Provides warnings for **obsolete features** and migration guidance.
- **Sanitizers** (ASan, UBSan, MSan): Fully compatible with C++23 code, including new memory safety and lifetime extensions.

2.4.5 Continuous Integration (CI) and Cross-Platform Support

- Major CI platforms (GitHub Actions, GitLab CI, Azure Pipelines) now provide **images with GCC/Clang/MSVC supporting C++23**.
- Cross-platform development benefits from **consistent C++23 compiler behavior across Linux, Windows, and macOS**.
- Embedded targets with ARM, RISC-V, and x86-64 toolchains require **careful verification of feature support**.

2.4.6 Recommendations for Professional Development

1. **Verify Compiler Version**
    - Ensure that your team uses **GCC 13+, Clang 16+, or MSVC 2022 v17.4+** for production-quality C++23 support.
2. **Use Modern Build Systems**
    - Adopt **CMake 3.26+** for consistent handling of C++23 features and cross-platform builds.
3. **Leverage IDE Capabilities**
    - Enable **static analysis, IntelliSense, and code refactoring** to avoid deprecated patterns.
4. **Test Legacy Code Migration**
    - Identify **removed or deprecated features** and refactor accordingly before enabling full C++23.
5. **Monitor Standard Library Updates**
    - Ensure that **libstdc++**, **libc++**, or MSVC STL are updated to support **all C++23 containers, ranges, and utilities**.

2.4.7 Conclusion

Tooling and compiler support are critical for **effective C++23 adoption**. While language and library features define the standard, **modern tooling ensures developers can use these features safely, efficiently, and consistently**.

By selecting the right **compilers, build systems, IDEs, and static analysis tools**, teams can fully leverage C++23's capabilities while maintaining **cross-platform compatibility and high performance**.

Part II

New Language Features in C++23

Chapter 3

Explicit Object Parameter (deducing `this`)

3.1 Tooling and Compiler Support

The **explicit object parameter**, or **deducing `this`**, is one of the **most significant language enhancements in C++23**. It addresses longstanding limitations in **member function design, ref-qualifier management, and generic programming**.

3.1.1 The Problem with Traditional `this`

In C++98 through C++20:

- Member functions implicitly receive a pointer or reference to the object through the **`this` pointer**.
- Managing **cv-qualifiers (`const`, `volatile`)** and **ref-qualifiers (`&`, `&&`)** often

required:

- Overloading multiple versions of the same function.
- Boilerplate code for **const and non-const objects**.
- Manual template specialization for generic member functions.

Example (pre-C++23):

```
struct Widget {
    void process() & { std::cout << "lvalue object\n"; }
    void process() && { std::cout << "rvalue object\n"; }
    void process() const & { std::cout << "const lvalue\n"; }
};
```

- Here, **three separate functions** are required to cover all common cases.
- As classes grow, this **adds complexity and duplication**.

3.1.2 The Goal of Deducing `this`

C++23 introduces the **ability to explicitly declare `this` as a function parameter**, allowing:

1. **Generic member functions** that automatically adapt to:
   - lvalue (`&`) or rvalue (`&&`) objects
   - const-qualified objects (`const`)
   - volatile-qualified objects (`volatile`)
2. **Reduction of boilerplate** by collapsing multiple overloads into a single function.

3. **Integration with templates and concepts**, enabling **fully generic member functions** without complex SFINAE or tag dispatching.

3.1.3 Example: Simplifying Member Functions

Using deducing `this`:

```
struct Widget {
    void print(this auto& self) {
        std::cout << "Widget object\n";
    }

    void print_const(this const auto& self) {
        std::cout << "Const Widget\n";
    }
};

Widget w;
w.print();          // lvalue
Widget{}.print();   // rvalue
const Widget cw;
cw.print_const();   // const object
```

- One function can replace **multiple overloaded versions**.
- The type of `self` is **deduced automatically**, providing **type safety and clarity**.

3.1.4 Benefits and Motivation

Expressiveness

- Makes the **object type explicit** in function signatures.
- Developers can **clearly see the intended cv/ref qualifiers** at the function level.

Generic Programming

- Enables **member functions to participate in template-based generic programming** without verbose specialization.
- Works seamlessly with **concepts and `requires` clauses**.

Safety and Consistency

- Prevents **silent misuse of cv/ref qualifiers**.
- Reduces **overload explosion**, minimizing **errors in maintenance and upgrades**.

Readability and Maintainability

- Collapses multiple overloads into one concise function.
- Improves **code readability** by making object qualification explicit.
- Easier to refactor and extend large classes with multiple member functions.

3.1.5 Real-World Motivation

Deducing `this` is particularly valuable for:

- **Libraries and frameworks**:

  - Reduces boilerplate in **containers, iterators, and utility classes**.
  - Enhances **generic algorithms inside class templates**.

- **Embedded and performance-critical systems**:
  - Ensures **const correctness and rvalue optimization** without additional overloads.

- **Modern APIs**:
  - Simplifies **fluent interfaces and chaining patterns** with minimal code duplication.

3.1.6 Summary

The **explicit object parameter (deducing `this`)** in C++23 addresses a fundamental limitation of earlier C++ standards: the implicit and rigid `this` pointer.

- Motivated by the need for **safer, more expressive, and maintainable member functions**.
- Enables **generic, const-correct, and ref-qualified member functions**.
- Reduces **boilerplate** and **overload complexity**, making large and generic codebases **easier to maintain**.

This feature forms a **cornerstone of modern C++23 design philosophy**, emphasizing **expressiveness, safety, and efficiency**, and sets the stage for **advanced usage with templates, concepts, and coroutines** in later chapters.

3.2 Syntax & Examples

C++23 introduces the **explicit object parameter**, which allows **member functions to declare `this` explicitly** as a function parameter. This feature is formally known as **deducing `this`** and enables **generic and expressive member function definitions**.

3.2.1 Basic Syntax

The general syntax for a **member function with deducing `this`** is:

```
return_type function_name(this parameter_type self, other_parameters...)
```

Where:

- `this parameter_type self` explicitly declares the **type of `this`**.
- `self` can be named anything (commonly `self`).
- `parameter_type` can include **cv-qualifiers (`const`, `volatile`) and reference qualifiers (`&`, `&&`)**.

```
struct Widget {
    void print(this auto& self) {
        std::cout << "This is a Widget object.\n";
    }
};

Widget w;
```

```
w.print();           // lvalue object
Widget{}.print();    // rvalue object
```

Example: Simple Deduction

- `auto&` deduces `this` as a reference to the object, automatically handling lvalues and rvalues when combined with **ref qualifiers**.

```
struct Widget {
    void show(this const auto& self) {
        std::cout << "Const object\n";
    }

    void move_show(this auto&& self) {
        std::cout << "Rvalue object\n";
    }
};

const Widget cw;
cw.show();           // const lvalue
Widget{}.move_show(); // rvalue
```

Const and Ref-Qualified Objects

- **`const auto&`** deduces `this` as a **const reference**, guaranteeing **read-only access**.

- **`auto&&`** deduces `this` as an **rvalue reference**, enabling **move semantics**.

3.2.2 Template Integration

Deducing `this` works seamlessly with **templated member functions**, allowing **generic member function definitions**.

```
struct Container {
    template<typename T>
    void append(this auto& self, const T& value) {
        self.data.push_back(value);
    }

private:
    std::vector<int> data;
};

Container c;
c.append(10);   // appends integer
```

- Templates combined with `this auto&` allow **generic member functions without explicit overloading**.

3.2.3 Concepts and `requires` Clauses

`this` can be used in **concept-constrained member functions**, enabling **conditional compilation based on the object type**.

```
#include <concepts>

struct Number {
    void double_value(this auto& self) requires std::integral<decltype(self.value)> {
        self.value *= 2;
```

```
    }

    int value{10};
};

Number n;
n.double_value(); // works because 'value' is integral
```

- Ensures **compile-time safety** and **type correctness**.

3.2.4 Rvalue and Lvalue Overload Reduction

Before C++23, multiple overloads were often needed:

```
struct Widget {
    void process() &;
    void process() &&;
    void process() const &;
};
```

With deducing `this`:

```
struct Widget {
    void process(this auto&& self) {
        if constexpr (std::is_lvalue_reference_v<decltype(self)>) {
            std::cout << "Lvalue\n";
        } else {
            std::cout << "Rvalue\n";
        }
    }
};
```

- One function handles **all cases**, reducing boilerplate and **improving maintainability**.

3.2.5 Constexpr and Deduction

Deducing `this` can be used in **constexpr functions**, enabling **compile-time evaluation**:

```
struct Widget {
    constexpr int get_value(this const auto& self) {
        return 42;
    }
};

constexpr Widget w;
static_assert(w.get_value() == 42);
```

- Allows **constant expressions**, aligning with **modern C++23 constexpr improvements**.

3.2.6 Best Practices

1. **Prefer `auto&` or `auto&&`** for maximum genericity.
2. **Use `const auto&`** when the function should not modify the object.
3. **Combine with templates or concepts** to maximize flexibility and safety.
4. **Minimize overloads** by leveraging deducing `this` instead of creating multiple ref-qualified versions.

5. Ensure **compiler support** (GCC 14+, Clang 17+, MSVC 2022 v17.5+) when using deducing `this`.

3.2.7 Summary

Deducing `this` in C++23 provides:

- **Clear, explicit control** over object type and cv/ref qualifiers.
- **Reduction of boilerplate code** and overloads.
- **Full integration with templates, concepts, and constexpr functions**.
- **Safer and more maintainable code**, enabling advanced generic programming patterns.

This section establishes the **syntax foundation** for deducing `this`, preparing for **Section 3**, where we will explore **Advanced Usage and Migration Strategies**, including **refactoring existing code** to leverage this feature efficiently.

3.3 Benefits for Generic Programming

Generic programming in C++ aims to **write code that works with any type** while maintaining **type safety and efficiency**. Traditional member function design limited the **expressiveness and reusability of member functions** due to the **implicit nature of `this`**, requiring multiple overloads and boilerplate code for **lvalue, rvalue, const, and volatile objects**.
C++23's **deducing `this`** addresses these limitations, providing **powerful benefits for generic programming**.

3.3.1 Unified Member Function for Multiple Object Types

Before C++23, handling multiple object categories required separate overloads:

```
struct Container {
    void update() & { /* lvalue */ }
    void update() && { /* rvalue */ }
    void update() const & { /* const lvalue */ }
};
```

- Three functions are needed to cover **lvalues, rvalues, and const objects**.
- This approach **increases maintenance complexity** as classes grow.

With deducing `this`:

```
struct Container {
    void update(this auto&& self) {
        if constexpr (std::is_lvalue_reference_v<decltype(self)>) {
            std::cout << "Lvalue\n";
```

```
        } else {
            std::cout << "Rvalue\n";
        }
    }
};
```

- One function handles **all object categories**.
- Ref-qualified behavior is **automatically deduced**, reducing boilerplate.

3.3.2 Integration with Templates

Deducing `this` works seamlessly with **template member functions**, enabling **generic operations on objects of various types**:

```
template<typename T>
struct Wrapper {
    template<typename U>
    void combine(this auto& self, const U& value) {
        self.data.push_back(value);
    }

private:
    std::vector<T> data;
};

Wrapper<int> w;
w.combine(10);        // integer
w.combine(20);        // integer
```

- Eliminates the need for **specializations or multiple overloads** for different `T` types.

- Works for **heterogeneous types**, as long as the operation is valid.

3.3.3 Concepts and Constraints

When combined with **concepts**, deducing `this` allows **fine-grained compile-time constraints**:

```
#include <concepts>

struct Number {
    void double_value(this auto& self) requires std::integral<decltype(self.value)> {
        self.value *= 2;
    }

    int value{10};
};

Number n;
n.double_value(); // works because 'value' is integral
```

- Only objects satisfying the concept (`std::integral`) can invoke the member function.
- Improves **compile-time safety** in generic libraries.

3.3.4 Support for Constexpr and Compile-Time Evaluation

Deducing `this` can be used in **constexpr member functions**, enabling **generic code evaluated at compile time**:

```
struct Widget {
    constexpr int value(this const auto& self) { return 42; }
};

constexpr Widget w;
static_assert(w.value() == 42);
```

- Generic member functions can now **operate on constant expressions**, which is crucial for **compile-time computations and metaprogramming**.

3.3.5 Code Reuse and Maintainability

- Reduces **function overload explosion**: one generic function can replace multiple ref-qualified versions.
- Simplifies **library design**, especially in **containers, iterators, and fluent APIs**.
- Enhances **maintainability**, as updates to the member function logic **propagate automatically** across all object categories.

Example in a generic **container API**:

```
template<typename Container>
struct Logger {
    void log(this auto& self, const std::string& message) {
        self.container.push_back(message);
    }

private:
    Container container;
```

```
};

Logger<std::vector<std::string>> logger;
logger.log("Event 1");
logger.log("Event 2");
```

- One function handles **any container type**, maintaining **generic behavior**.

3.3.6 Motivation for Library Developers

For library developers, deducing `this`:

1. **Simplifies API design**: Avoids multiple overloads for different cv/ref combinations.
2. **Enables generic, reusable components**: Works across lvalues, rvalues, and const-qualified objects.
3. **Supports modern C++ patterns**: Integrates with **concepts, ranges, coroutines, and constexpr**.
4. **Improves code clarity**: Makes **object qualification explicit**, reducing ambiguity.

This is particularly valuable for:

- **Generic containers**
- **Utility libraries**
- **Fluent APIs and chainable functions**
- **Performance-critical systems** where **move semantics and const correctness** are essential

3.3.7 Summary

Deducing `this` in C++23 **unlocks new potential for generic programming**:

- **One member function can handle multiple object categories** (lvalue, rvalue, const, volatile).
- Works seamlessly with **templates and concepts** to produce **safe, expressive, and reusable code**.
- **Reduces boilerplate and overloads**, improving maintainability and readability.
- Enables **compile-time evaluation in constexpr contexts**, enhancing metaprogramming capabilities.

In modern C++23, **deducing `this` is an essential tool for writing generic, type-safe, and maintainable libraries**, making it a **cornerstone of advanced C++ programming**.

Chapter 4

Multidimensional Subscript Operator

4.1 `v[1,2,3]` Syntax

C++23 introduces the **ability to use multiple subscripts within a single pair of square brackets**, enabling a **natural and concise syntax for multidimensional indexing**. This feature is commonly referred to as the **multidimensional subscript operator**.

4.1.1 Motivation

Before C++23:

- Accessing elements of multidimensional structures required **nested square brackets**:

```
int matrix[3][3][3];
matrix[0][1][2] = 42;
```

- For **custom containers**, supporting multidimensional indexing often required **complex nested operator[] implementations**.
- The nested syntax could become **verbose, error-prone, and less readable** in high-dimensional arrays or containers.

C++23 introduces **a simplified, comma-separated indexing syntax**, making **multidimensional access more natural**.

4.1.2 Syntax

The **new syntax** allows multiple indices separated by commas inside a **single pair of brackets**:

```
container[i, j, k]
```

Where:

- `container` is a **user-defined type** that overloads `operator[]`.
- Each index corresponds to **a dimension** in the container.

The **comma operator** inside the brackets is **overloadable for custom containers**, allowing **user-defined multidimensional behavior**.

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

struct Matrix3D {
    int data[3][3][3];

    int& operator[](std::tuple<int,int,int> indices) {
        auto [i, j, k] = indices;
        return data[i][j][k];
    }
};

int main() {
    Matrix3D mat{};
    mat[std::make_tuple(0,1,2)] = 42;   // traditional approach

    // C++23 simplified syntax
    // mat[0,1,2] = 42;  // enabled if operator[] supports comma-overload
}
```

Basic Example with std::array

- Standard containers do not yet natively support `v[1,2,3]`, but **user-defined containers can implement it**.
- The syntax enables **concise multidimensional access** without nested brackets.

4.1.3 Overloading the Multidimensional Subscript

To support `v[1,2,3]`, **custom containers overload `operator[]`** to accept a **parameter pack** or **tuple of indices**:

```
struct Grid3D {
    int data[3][3][3];

    template<typename... Indices>
    int& operator[](Indices... idx) {
        static_assert(sizeof...(idx) == 3, "Requires exactly 3 indices");
        auto indices = std::array<int, sizeof...(idx)>{idx...};
        return data[indices[0]][indices[1]][indices[2]];
    }
};

Grid3D grid;
grid[0,1,2] = 100;
std::cout << grid[0,1,2];  // prints 100
```

- The **parameter pack** allows for **arbitrary dimensional indexing**.
- `static_assert` ensures **compile-time safety** for the number of dimensions.

4.1.4 Practical Benefits

1. **Readable Code**
    - One pair of brackets replaces **nested indexing**, improving readability.

```
grid[0,1,2];    // vs
grid[0][1][2];
```

1. **Custom Containers Support**

- Library authors can design **multidimensional containers** that behave naturally with `v[1,2,3]`.

2. **Integration with Ranges and Views**

   - Works with **C++23 ranges** to create **powerful multidimensional algorithms**.

3. **Consistency with Python/Matlab Style**

   - Provides a **familiar multidimensional indexing syntax** for developers coming from **dynamic languages**.

4.1.5 Safety and Best Practices

- Always **validate indices** inside the overloaded `operator[]` to avoid undefined behavior.
- Consider **constexpr operator**[] for compile-time evaluation of fixed-size grids.
- Use **parameter packs with static_assert** to enforce **dimension correctness** at compile time.
- Avoid using the syntax with types that cannot naturally support it (like `std::vector` without wrapper classes).

4.1.6 Summary

The **multidimensional subscript operator (`v[1,2,3]`)** in C++23:

- Simplifies **access to multidimensional data structures**.

- Reduces **nested bracket verbosity** and enhances **readability**.
- Allows **user-defined containers** to provide **flexible, safe, and generic indexing**.
- Works seamlessly with **parameter packs, tuples, and constexpr functions**.

This feature **paves the way for modern multidimensional data structures** in libraries and applications, aligning C++23 with **high-level indexing capabilities seen in other languages** while maintaining **compile-time safety and performance**.

4.2 Overload Rules

C++23 introduces the **multidimensional subscript operator**, allowing multiple indices in a single pair of brackets. This feature **relies on overloading the comma operator inside the context of `operator[]`**, enabling **custom handling for each dimension**. Understanding **overload rules** is critical to ensure **correct, predictable behavior** in multidimensional containers.

4.2.1 Basic Principles

1. **Operator Overload Hierarchy**

    - The **multidimensional subscript operator** is implemented by **overloading `operator[]`** in the container class.
    - Each call to `operator[]` may **return a proxy object**, which can **chain further indexing** or resolve the final element.

2. **Comma Interpretation**

    - Inside the brackets, commas **normally invoke the built-in comma operator**, which evaluates the left-hand expression and discards it, returning the right-hand expression.
    - C++23 allows **overloading this behavior** for multidimensional indexing **through templates, tuples, or parameter packs**.

4.2.2 Overload Resolution Rules

When multiple `operator[]` overloads exist, C++23 uses the **same overload resolution rules as any member function**:

1. **Exact Match Preference**
    - An `operator[]` accepting **exact types** of indices is preferred over template-based overloads.
2. **Template Deduction**
    - If no exact match exists, **templated `operator[]`** is considered, with **type deduction** for parameter packs or tuples.
3. **Const-Qualified Objects**
    - Overloads for **const objects** take priority when the object is const.
    - Example:

```
struct Grid {
    int& operator[](std::tuple<int,int,int> idx);           // non-const
    int operator[](std::tuple<int,int,int> idx) const;     // const
};
```

- **`const Grid g; g[{1,2,3}]`** calls the const version automatically.

1. **Lvalue vs Rvalue Reference Qualification**
    - `operator[]` can be **ref-qualified** (`&` or `&&`) to distinguish between lvalues and rvalues.
    - This allows **move semantics or temporary-specific behavior**:

```
struct Matrix {
    int& operator[](std::array<int,3> idx) &;   // lvalue
    int&& operator[](std::array<int,3> idx) &&; // rvalue
};
```

- C++23 allows **combined comma-indexing with ref-qualified overloads**, improving flexibility.

4.2.3 Using Parameter Packs

A common implementation uses **template parameter packs** to capture multiple indices:

```
struct Grid3D {
    int data[3][3][3];

    template<typename... Indices>
    int& operator[](Indices... idx) {
        static_assert(sizeof...(idx) == 3, "Exactly 3 indices required");
        auto indices = std::array<int, sizeof...(idx)>{idx...};
        return data[indices[0]][indices[1]][indices[2]];
    }
};
```

Rules:

1. **Number of indices must match the dimensionality**.
2. **Types must be convertible** to the expected index type (`int`, `size_t`, etc.).
3. **Ref-qualifiers can be combined** to distinguish lvalue/rvalue objects.

4.2.4 Interaction with Built-in Comma Operator

- **C++23 allows overloading comma behavior in `operator[]` contexts**.
- For standard containers, the **comma is still evaluated as the built-in operator**, but in **user-defined containers**, you can interpret the indices collectively.

Example with multiple dimensions using a single overload:

```
struct Grid3D {
    int data[3][3][3];

    template<typename I1, typename I2, typename I3>
    int& operator[](std::tuple<I1,I2,I3> indices) {
        auto [i,j,k] = indices;
        return data[i][j][k];
    }
};
```

- This ensures **predictable indexing** while avoiding unintended evaluation from the built-in comma operator.

4.2.5 Constexpr and Compile-Time Overloads

- Overloading `operator[]` for **constexpr containers** enables **compile-time multidimensional access**:

```
struct ConstexprGrid {
    int data[2][2][2]{};
```

```
    template<typename... Indices>
    constexpr int& operator[](Indices... idx) {
        static_assert(sizeof...(idx) == 3, "Requires 3 indices");
        auto indices = std::array<int, sizeof...(idx)>{idx...};
        return data[indices[0]][indices[1]][indices[2]];
    }
};

constexpr ConstexprGrid g{};
static_assert(g[0,1,1] == 0);  // compile-time check
```

- Ensures **high-performance, zero-overhead multidimensional indexing** for fixed-size grids.

4.2.6 Best Practices for Overloads

1. **Validate index count and types** using `static_assert`.
2. **Use parameter packs or tuples** for clean, scalable overloads.
3. **Ref-qualify operator[]** to handle lvalues and rvalues differently.
4. **Provide const and non-const overloads** for safety.
5. **Document expected behavior** when overloading comma behavior inside operator[].

4.2.7 Summary

The **overload rules for the multidimensional subscript operator** in C++23 provide:

- Flexibility to **accept multiple indices** in a single bracket pair.
- Compile-time **dimension and type safety** using static_asserts and template packs.
- Seamless support for **const, non-const, lvalue, and rvalue objects**.
- Control over **comma operator evaluation**, ensuring consistent behavior.
- Integration with **constexpr and compile-time multidimensional arrays**.

These rules **empower library developers** to implement **efficient, safe, and readable multidimensional containers**, while maintaining **full compatibility with existing C++ idioms**.

4.3 Use in Math/Scientific Programming

Mathematical, scientific, and engineering applications frequently rely on **multidimensional arrays, matrices, tensors, and higher-dimensional data structures**. Traditional C++ approaches often require **nested loops and nested indexing**, which can be **verbose, error-prone, and less readable**.
C++23's **multidimensional subscript operator** (`v[1,2,3]`) significantly simplifies **access, manipulation, and readability** for such applications.

4.3.1 Matrix and Tensor Access

```
double matrix[3][3][3];
matrix[0][1][2] = 42.0;

for(int i=0; i<3; ++i)
    for(int j=0; j<3; ++j)
        for(int k=0; k<3; ++k)
            matrix[i][j][k] = i + j + k;
```

Traditional Approach (Pre-C++23):

- Nested brackets and loops **reduce code clarity**.
- Adding **custom containers** required **overloading multiple `operator[]`** for each dimension.

```
struct Tensor3D {
    double data[3][3][3];
```

```
    template<typename... Indices>
    double& operator[](Indices... idx) {
        static_assert(sizeof...(idx) == 3, "Requires 3 indices");
        auto indices = std::array<int, sizeof...(idx)>{idx...};
        return data[indices[0]][indices[1]][indices[2]];
    }
};

Tensor3D tensor;
tensor[0,1,2] = 42.0;       // concise
tensor[1,2,0] = 5.0;
```

C++23 Multidimensional Subscript Operator:

- One **overloaded operator** handles all indices.
- Readable, concise, and easy to maintain.

4.3.2 Linear Algebra Applications

- Scientific libraries often define **matrices and vectors**.
- The new syntax enables **clearer mathematical expressions**:

```
struct Matrix3x3 {
    double data[3][3];

    template<typename I1, typename I2>
    double& operator[](std::tuple<I1,I2> indices) {
        auto [i,j] = indices;
        return data[i][j];
```

```
    }
};

Matrix3x3 m;
m[std::make_tuple(0,2)] = 1.5; // old way
// m[0,2] = 1.5; // C++23 simplified syntax (if operator[] allows comma overloading)
```

- Supports **vectorized algorithms**, **matrix multiplication**, and **linear transformations** with minimal code clutter.

4.3.3 Tensors for Scientific Simulations

High-dimensional data structures are essential in:

- Physics simulations (3D/4D grids, lattice computations)
- Computational chemistry (molecular modeling, electron densities)
- Machine learning (multi-channel data arrays, tensors)

```
struct Tensor4D {
    double data[2][3][3][3];

    template<typename... Indices>
    double& operator[](Indices... idx) {
        static_assert(sizeof...(idx) == 4, "Requires 4 indices");
        auto indices = std::array<int, sizeof...(idx)>{idx...};
        return data[indices[0]][indices[1]][indices[2]][indices[3]];
    }
};
```

```
Tensor4D t;
t[1,2,0,2] = 3.1415;  // concise, clear indexing
```

Example: 4D Tensor

- Eliminates **nested bracket verbosity**.
- Supports **compile-time dimension checks** via `static_assert`.

4.3.4 Integration with Numerical Libraries

Custom containers using multidimensional subscript operators can integrate with:

- **Eigen-style libraries**: for matrix and vector algebra
- **BLAS/LAPACK bindings**: for linear algebra operations
- **Finite element libraries**: for multidimensional mesh grids
- **Simulation engines**: for multidimensional physical models

Example: Custom wrapper over `std::vector` for arbitrary dimensions:

```
template<typename T, size_t Dim1, size_t Dim2, size_t Dim3>
struct Matrix {
    std::vector<T> data{Dim1*Dim2*Dim3};

    T& operator[](std::tuple<size_t,size_t,size_t> idx) {
        auto [i,j,k] = idx;
        return data[i*Dim2*Dim3 + j*Dim3 + k];
    }
};
```

```
Matrix<double,3,3,3> mat;
mat[std::make_tuple(0,1,2)] = 9.8;
```

- Encapsulates **flat storage** while providing **natural multidimensional access**.
- Reduces **memory layout complexity** without sacrificing performance.

4.3.5 Benefits for Math/Scientific Programming

1. **Readable Code**: Single bracket syntax improves clarity for multidimensional arrays.
2. **Compile-Time Safety**: `static_assert` ensures correct number of dimensions.
3. **Reduced Boilerplate**: Eliminates the need for multiple nested `operator[]` or proxy classes.
4. **High Performance**: Enables **zero-overhead access** in fixed-size containers or **optimized flat storage arrays**.
5. **Generic Algorithms**: Works naturally with **template-based linear algebra routines**.

4.3.6 Best Practices

- Always **validate indices** to avoid out-of-bounds access.
- Use **constexpr for fixed-size arrays** in compile-time algorithms.
- Consider **flattened storage** for performance-critical applications while providing **multidimensional syntax for access**.

- Combine with **ranges, concepts, and generic algorithms** to write **expressive, high-level numerical code**.

4.3.7 Summary

The multidimensional subscript operator in C++23:

- Enables **concise and readable access to multidimensional arrays, matrices, and tensors**.
- Supports **compile-time dimension validation**, improving safety in scientific computations.
- Reduces boilerplate in **math-heavy applications** such as linear algebra, simulations, and tensor computations.
- Aligns C++ with **modern scientific languages** in terms of **natural, expressive indexing syntax** while maintaining **performance and type safety**.

Chapter 5

Static Callables

5.1 `static operator[]`, `static operator()`, and `static lambdas`

C++23 introduces the ability to define **static callable members**, including **static `operator[]`, static `operator()`, and static lambda expressions**. These features enable **callable behaviors that do not require object instantiation**, improving **efficiency, clarity, and design flexibility**.

5.1.1 Motivation

Traditionally in C++:

- **Operators like `operator[]` and `operator()`** could only be non-static for object instances.
- **Lambdas** captured state implicitly or required external variables, making **purely static computations more verbose**.

- Developers often had to rely on **free functions or static utility classes** for stateless callable behavior.

C++23 solves this by allowing **callable members to be static**, enabling:

- **Direct class-level invocations** without creating an object.
- **State-free, high-performance utilities**.
- Cleaner syntax for **generic and functional programming patterns**.

5.1.2 Static `operator()`

```
struct Math {
    static int operator()(int x, int y) {
        return x + y;
    }
};

int result = Math::operator()(5, 7); // returns 12
```

Syntax

- `operator()` can now be **invoked directly on the class**.
- Eliminates the need to **instantiate an object** for stateless operations.
- Useful for **stateless function objects** and **compile-time utilities**.

```
struct Factorial {
    static constexpr int operator()(int n) {
        return n <= 1 ? 1 : n * Factorial{}(n-1);
    }
};

constexpr int val = Factorial::operator()(5); // 120
```

Example: Compile-Time Computation

- Supports **constexpr evaluation**, enabling **compile-time calculations**.
- Reduces **overhead** by avoiding object instantiation.

5.1.3 Static `operator[]`

```
struct Lookup {
    static int operator[](int index) {
        constexpr int table[] = {10, 20, 30, 40};
        return table[index];
    }
};

int val = Lookup::operator ; // returns 30
```

Syntax

- Enables **array-like access directly from the class**.
- Useful for **precomputed lookup tables** or **static configuration arrays**.

Notes:

- Indexing behavior is **entirely static**, so **no object state** is needed.
- Can be combined with **constexpr** for **compile-time table lookups**.

5.1.4 Static Lambdas

C++23 allows **lambdas to be declared static**, which means:

- They **cannot capture instance variables**.
- They **can be called without creating an instance**.
- They are **ideal for generic, stateless, or utility operations**.

```
static auto sum = [](int a, int b) -> int {
    return a + b;
};

int result = sum(3, 4); // 7
```

Syntax

- Can also be **used inside classes**:

```
struct Utils {
    static auto multiply = [](int x, int y) { return x * y; };
};

int result = Utils::multiply(6, 7); // 42
```

- Provides **concise syntax for stateless function objects**.

5.1.5 Advantages of Static Callables

1. **No Object Instantiation**
    - Reduces memory and runtime overhead for stateless operations.
2. **Compile-Time Evaluation**
    - Combined with `constexpr`, allows **compile-time computation** and optimizations.
3. **Improved Generic Programming**
    - Can be passed as template parameters or used in **higher-order functions** without creating objects.
4. **Cleaner API Design**
    - Library authors can expose **class-level operations** without unnecessary object instances.
5. **Integration with Algorithms**
    - Works seamlessly with **ranges, algorithms, and functional patterns** in C++23.

5.1.6 Best Practices

- Use **static callables** for **stateless computations** or **utility operations**.
- Prefer **constexpr static callables** for **compile-time evaluation**.

- Avoid **capturing instance state** in static lambdas; they are designed to be **self-contained**.
- Combine **static operators with templates** for **generic, high-performance algorithms**.

5.1.7 Summary

C++23's **static callables** feature, including **`static operator[]`, `static operator()`, and `static lambdas`**, provides:

- **Direct class-level invocation** without object creation.
- Enhanced **readability and expressiveness** for stateless operations.
- Support for **compile-time evaluation** with `constexpr`.
- Improved **generic programming capabilities**, allowing functions and algorithms to operate on **stateless callables efficiently**.
- Simplification of **utility classes, mathematical libraries, and lookup structures**.

This feature brings C++ closer to **functional programming paradigms** while maintaining **high performance and type safety**.

5.2 Examples in Generic Programming

Static callables (`static operator()`, `static operator[]`, `static lambdas`) are particularly powerful in **generic programming** because they:

- Do not require object instantiation, making them ideal for **template parameters**.
- Can be evaluated at **compile-time** with `constexpr`.
- Support **highly reusable and stateless algorithms**.

Below we explore practical examples.

1. Static Callable as Template Parameter Templates in C++ often accept **functors** or **function objects**. Static callables simplify this by **eliminating the need to instantiate the functor**.

```
template<typename Callable, typename T>
constexpr T apply(T value) {
    return Callable::operator()(value);
}

// Define static callable
struct Square {
    static constexpr int operator()(int x) { return x * x; }
};

constexpr int result = apply<Square>(5);  // 25
```

Advantages:

- `Square` is **never instantiated**.
- Fully **constexpr-capable**, enabling **compile-time evaluation**.
- Works with **generic templates** for various data types.

2. Static Lambdas with Templates C++23 allows **static lambdas** to be used directly in template contexts.

```
template<auto Func, typename T>
constexpr T compute(T value) {
    return Func(value);
}

// Static lambda
constexpr auto cube = [](int x) { return x * x * x; };

constexpr int result = compute<cube>(3);  // 27
```

- The lambda is **treated as a static, type-safe callable**.
- No need for object creation; usable as a **non-type template parameter**.

3. Static `operator[]` in Generic Containers Generic containers or mathematical structures can leverage **static `operator[]`** for **compile-time lookup tables**.

```
template<typename T, size_t N>
struct LookupTable {
    static constexpr T values[N] = {1, 2, 4, 8, 16};

    static constexpr T operator[](size_t index) {
        return values[index];
```

```
    }
};

constexpr int val = LookupTable<int, 5>::operator ;  // 8
```

Benefits for generic programming:

- Works with **any type `T`**.
- Supports **compile-time indexing**.
- Can be passed as a **template argument** to algorithms or utilities.

4. Static Callables for Higher-Order Functions Higher-order functions accept other functions as parameters. Static callables make **generic, stateless operations simpler and safer**.

```
template<typename Func, typename T>
T transform(T value) {
    return Func::operator()(value);
}

struct Double {
    static int operator()(int x) { return x * 2; }
};

int result = transform<Double>(10); // 20
```

- `Double` can be **swapped with any static callable**.
- Avoids runtime overhead of object instantiation.

5. Combining Static Callables with Concepts C++23 allows static callables to **work naturally with concepts**, enforcing **constraints at compile-time**.

```
template<typename Func, typename T>
requires requires(T x) { Func::operator()(x); }
T apply_generic(T value) {
    return Func::operator()(value);
}

struct Increment {
    static int operator()(int x) { return x + 1; }
};

int val = apply_generic<Increment>(5);  // 6
```

- Guarantees **correct usage at compile-time**.
- Supports **stateless, reusable components** in generic algorithms.

6. Use in Ranges and Algorithms Static callables integrate well with **C++23 ranges and algorithms**:

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

struct Square {
    static int operator()(int x) { return x * x; }
};

std::vector<int> data = {1, 2, 3, 4, 5};
auto squared = data | std::views::transform(Square{}); // static callable passed as
↪  functor
```

- Enables **generic transformation pipelines**.
- Works without **creating additional stateful objects**.

7. Best Practices for Generic Programming

1. Use **static callables** when **state is not needed**.
2. Prefer **`constexpr` static callables** for **compile-time evaluation**.
3. Combine with **templates and concepts** to create **robust, reusable algorithms**.
4. Use **static lambdas** to write **concise, inline stateless logic**.
5. Integrate with **ranges and modern algorithms** for expressive, high-level code.

8. Summary Static callables in C++23 provide:

- **Zero-overhead stateless callables** suitable for **generic programming**.
- Support for **template-based algorithms, higher-order functions, and compile-time computation**.
- Seamless integration with **concepts, ranges, and algorithms**.
- Simplified, readable, and **reusable code patterns** across math, utility, and library development.

These features **enhance C++23's expressive power** while maintaining **high performance, safety, and clarity** in generic programming contexts.

Chapter 6

Decay-Copy with `auto(x)` and `auto{x}`

6.1 Subtle Differences in Type Deduction

C++23 introduces **decay-copy syntax for `auto(x)` and `auto{x}`**, which enables **more predictable and consistent type deduction** when initializing variables with temporary expressions. This feature resolves some **ambiguities from prior standards** regarding **references, arrays, and function pointers**.

6.1.1 Motivation

In previous C++ versions:

```
int a = 42;
auto x = a;       // x is int
auto& y = a;      // y is int&
```

```
int arr[3] = {1,2,3};
auto b = arr;     // b is int*, not int[3], due to array decay
```

- Arrays **decay into pointers** automatically.
- Function types **decay into function pointers**.
- Template code can behave inconsistently due to **unexpected decay**.

C++23 introduces **explicit decay-copy syntax** to give developers **control over this behavior**.

6.1.2 `auto(x)` Syntax

- `auto(x)` performs a **decay-copy of the initializer `x`**.
- Guarantees the **type is decayed** (arrays → pointer, function → pointer) **before deducing `auto`**.
- Ensures **value semantics** rather than reference semantics.

```
int arr[3] = {1, 2, 3};
auto a = arr;      // a is int*, traditional decay
auto b = auto(arr); // b is also int*, explicit decay-copy
```

Example: Array Decay

```
void foo() {}

auto fptr1 = foo;       // fptr1 is void(*)()
auto fptr2 = auto(foo); // fptr2 is also void(*)(), decay-copy explicit
```

Example: Function Pointer

- `auto(x)` makes **decay explicit**, removing ambiguity in template contexts.

6.1.3 `auto{x}` Syntax

- `auto{x}` behaves like **direct-list-initialization**, including **braced-init-list rules**.
- Provides **uniform initialization semantics** and prevents **narrowing conversions**.
- Often used in **template code** to ensure **consistent type deduction for uniform initialization**.

```
double pi = 3.1415;
auto val1 = pi;      // val1 is double
auto val2 = auto{pi}; // val2 is double, prevents accidental narrowing if used with
↪  smaller types
```

Example: Narrowing Conversion

```
int arr[3] = {1,2,3};
auto b = auto{arr};  // b is int*, consistent decay-copy
```

Example: Braced Initialization

- Guarantees **safe initialization** in generic contexts.

6.1.4 Subtle Differences

Feature	`auto(x)`	`auto{x}`
Array initialization	Decays to pointer	Same, braced-init semantics
Function type	Decays to function pointer	Same
Reference binding	Decays to value	Prevents implicit narrowing
Template consistency	Ensures decay-copy	Uniform braced initialization
Narrowing checks	No	Yes

- `auto{x}` is safer for **generic code** where **narrowing conversions are undesirable**.
- `auto(x)` is convenient for **classic decay-copy scenarios**.

6.1.5 Implications for Generic Programming

1. **Templates become more predictable**

- Decay-copy ensures **arrays, functions, and other non-class types** behave consistently in templates.

2. **Supports compile-time safety**

   - Avoids unintended type conversions or reference binding.

3. **Integration with `constexpr` and static callables**

   - Decay-copy ensures **values passed to static callables or constexpr functions** have consistent types.

4. **Readable and maintainable code**

   - Reduces **surprises in type deduction** when migrating old code to modern C++ standards.

6.1.6 Best Practices

- Use `auto(x)` when **classic decay-copy semantics** are desired.
- Use `auto{x}` when **uniform initialization and narrowing protection** are needed.
- Combine with **concepts and templates** for **predictable generic code**.
- Review old code relying on **implicit array or function decay**, and consider **explicit decay-copy** to prevent subtle bugs.

6.1.7 Summary

C++23's **decay-copy syntax with `auto(x)` and `auto{x}`**:

- Provides **clear and consistent type deduction** for arrays, functions, and temporaries.
- Resolves **ambiguities present in prior standards**.
- Enhances **template and generic programming safety**.
- Supports **compile-time evaluation and static analysis**.
- Offers **predictable and maintainable initialization semantics** for modern C++ development.

This subtle yet powerful feature is essential for **C++ developers writing generic libraries, mathematical algorithms, and high-performance code**, where **type safety and predictability are critical**.

Chapter 7: New Attributes in C++23

Section 1: `[[assume]]` and Lambda Attributes

This section will provide a **detailed explanation of the new attributes introduced in C++23**, including **syntax, semantics, and examples**, which improve **program safety, optimization, and expressiveness**.

Chapter 7

New Attributes in C++23

7.1 `[[assume]]` and Lambda Attributes

C++23 introduces **new attributes** to enhance **compiler optimization, code safety, and clarity**. Among the most significant are:

1. `[[assume(expression)]]`
2. **Attributes on lambda expressions**

These attributes provide **compile-time guidance to the compiler** and improve **program readability and safety**.

7.1.1 Motivation

Attributes in C++ are **metadata** that convey **additional information to the compiler** without changing program semantics.
Prior to C++23:

- Attributes like `[[nodiscard]]`, `[[likely]]`, and `[[unlikely]]` were available.
- There was no direct way to **inform the compiler about assumptions regarding program state**, which could help with **optimization** and **static analysis**.
- Lambda expressions could not carry **additional metadata**, limiting **diagnostics and behavior hints**.

C++23 introduces:

- **`[[assume(expression)]]`** for **compiler-guided assumptions**.
- **Lambda attributes** to improve **diagnostics, compile-time checking, and generic programming**.

`[[assume(expression)]]`

```
[[assume(expression)]];
```

Syntax

- `expression` must be **evaluatable to `bool`** at compile-time or runtime.
- Communicates to the compiler: "**assume this condition is true** at this point in the program."

```
int divide(int x, int y) {
    [[assume(y != 0)]];
    return x / y;
}
```

Example: Optimization Hint

- The compiler can **omit checks for `y != 0`** if optimization allows.
- **Undefined behavior** if the assumption is violated; use carefully.

7.1.2 Benefits

1. **Enables aggressive optimizations**
    - Compilers can **skip redundant runtime checks**.
2. **Improves static analysis**
    - Tools can reason about program flow with known assumptions.
3. **Readable intent**
    - Makes **developer assumptions explicit** for future maintainers.

```
void process(int* data, size_t n) {
    [[assume(n % 4 == 0)]];
    for (size_t i = 0; i < n; i += 4) {
        // optimized loop assuming n divisible by 4
    }
}
```

Example: Loop Optimization

- Compiler can **vectorize loops** safely.
- Reduces runtime overhead while **maintaining explicit developer guarantees**.

7.1.3 Attributes on Lambda Expressions

Syntax C++23 allows **attributes to be applied directly to lambda expressions**:

```
auto lambda = [[nodiscard]] [](int x) { return x * 2; };
```

- Attributes are applied **inline with lambda definition**.
- Can specify **nodiscard, likely/unlikely, constexpr, or custom attributes**.

```
auto compute = [[nodiscard]] [](int x, int y) -> int {
    return x + y;
};

int result = compute(3, 4); // compiler warns if result ignored
```

Example: `nodiscard` with Lambda

- Improves **code safety by enforcing usage of results**.
- Integrates seamlessly with **generic programming and higher-order functions**.

```
auto square = [[nodiscard, constexpr]] [](int x) { return x * x; };

constexpr int val = square(5); // 25, evaluated at compile-time
```

Example: `constexpr` Lambda Attribute

- Lambda attributes make **compile-time evaluation and diagnostics explicit**.
- Enhances **integration with static analysis, templates, and generic code**.

7.1.4 Benefits of New Attributes

1. **Expressive and Readable Code**
   - Makes **intent clear** to both compiler and human readers.
2. **Safer Programming Practices**
   - `[[nodiscard]]` on lambdas prevents **ignored results**.
   - `[[assume]]` reduces unnecessary runtime checks while providing **developer assurances**.
3. **Better Compiler Optimization**
   - `[[assume]]` allows **vectorization, branch prediction hints, and loop optimizations**.
4. **Improved Generic and Template Programming**
   - Lambda attributes can **propagate attributes** in **higher-order functions**.
   - Enables **stateless, constexpr, and constrained lambdas** with explicit metadata.

7.1.5 Best Practices

- Use `[[assume(expression)]]` **only when you are certain** the expression is true. Violating the assumption leads to **undefined behavior**.
- Apply attributes to lambdas to **enforce safety (`nodiscard`) or constexpr behavior**.
- Combine with **static analysis tools** for **compile-time verification of assumptions**.
- Use attributes to **document design intent** clearly for future maintainers.

7.1.6 Summary

C++23 introduces **powerful new attributes** to enhance **code safety, optimization, and expressiveness**:

- `[[assume(expression)]]` enables **compiler-guided assumptions**, reducing runtime checks and improving optimization.
- Attributes on lambdas allow **inline metadata**, supporting **constexpr evaluation, result checking, and static analysis**.
- These attributes improve **generic programming, template safety, and maintainable code design**.

C++23's attribute system empowers developers to **write safer, more efficient, and more expressive modern C++**.

7.2 Correct and Incorrect Uses

C++23 introduces **new attributes** to improve **code safety, optimization, and expressiveness**. However, improper usage can lead to **undefined behavior or compiler warnings**. This section clarifies **what constitutes correct vs. incorrect usage**.

7.2.1 Correct Uses of `[[assume(expression)]]`

Precondition Assertions Use `[[assume]]` to **inform the compiler of conditions that are guaranteed to be true** at runtime:

```
int divide(int x, int y) {
    [[assume(y != 0)]];
    return x / y;
}
```

- Correct because **the caller must guarantee `y != 0`**.
- Compiler can optimize by **removing runtime checks**.

Loop Optimization When a loop has a known property that can aid **vectorization or unrolling**:

```
void process(int* data, size_t n) {
    [[assume(n % 4 == 0)]];
    for (size_t i = 0; i < n; i += 4) {
        // safe optimized loop
    }
}
```

- Correct use: `n` must indeed be divisible by 4 whenever the function is called.
- Helps compiler **generate more efficient machine code**.

Compile-Time Guarantees Use with **`constexpr` expressions** when **assumptions are known at compile time**:

```
constexpr int factorial(int n) {
    [[assume(n >= 0)]];
    int result = 1;
    for (int i = 1; i <= n; ++i) result *= i;
    return result;
}
```

- Correct because `n >= 0` is a **logical precondition**.
- Compiler may use this to **validate code paths** or optimize loops.

7.2.2 Incorrect Uses of `[[assume(expression)]]`

```
int divide(int x, int y) {
    [[assume(y != 0)]];
    return x / y;
}

int result = divide(5, 0); // undefined behavior
```

Violating the Assumption

- **Incorrect** because the assumption may be violated.
- Leads to **undefined behavior**; the compiler may omit runtime checks.

```
int divide(int x, int y) {
    [[assume(y != 0)]];   // INCORRECT if used as a runtime check
    return x / y;
}
```

Misuse as Assertion Replacement

- `[[assume]]` **does not throw errors** if violated.
- Use **assertions** (`assert(y != 0);`) for runtime validation instead.

```
void process(size_t n) {
    [[assume(n % 4 == 0)]];  // INCORRECT if n is dynamic and not guaranteed
    for (size_t i = 0; i < n; i += 4) { /* ... */ }
}
```

Dynamic Values Without Guarantees

- Safe only if **all callers guarantee the condition**.
- Otherwise, leads to **compiler optimizations that assume false preconditions**.

7.2.3 Correct Uses of Lambda Attributes

```
auto compute = [[nodiscard]] [](int x, int y) { return x + y; };
int result = compute(3, 4); // Correct: result used
```

Enforcing Result Usage

- Correct because **the lambda result is used**.
- Compiler can warn if the result is ignored:

```
compute(3, 4); // Warning: discard of [[nodiscard]] value
```

```
auto square = [[nodiscard, constexpr]] [](int x) { return x * x; };
constexpr int val = square(5); // correct, compile-time evaluation
```

`constexpr` Lambdas

- Correct because lambda **can be evaluated at compile-time**.

7.2.4 Incorrect Uses of Lambda Attributes

```
auto compute = [[nodiscard]] [](int x, int y) { return x + y; };
compute(3, 4); // Warning: discard of nodiscard value
```

Ignored `[[nodiscard]]` Value

- Incorrect because the **result is ignored**, negating the attribute's purpose.

```
[[nodiscard]] int value = 42; // correct
[[nodiscard]] auto func = 42; // incorrect if not a callable or result type
```

Applying Attributes to Non-Lambdas

- Some attributes are **meaningful only on functions/lambdas**.

7.2.5 Guidelines for Correct Usage

1. **`[[assume]]`**
    - Use only when **conditions are guaranteed**.
    - Do not rely on it as a **runtime check**.
    - Combine with **assert** for dynamic safety.
2. **Lambda Attributes**
    - Apply `[[nodiscard]]` to enforce **result usage**.
    - Apply `constexpr` or `consteval` for **compile-time lambdas**.
    - Avoid applying attributes that do not match the **lambda type or context**.
3. **Document Assumptions**
    - Clearly state assumptions for `[[assume]]` in **comments or documentation**.
    - Improves maintainability and **avoids misuse by future developers**.

7.2.6 Summary

Correct and incorrect usage of **C++23 attributes** is critical for:

- **Safe optimization with `[[assume]]`**
- **Enforcing result usage and compile-time evaluation with lambda attributes**
- Preventing **undefined behavior** and compiler warnings

C++23 attributes provide **powerful tools** but require **careful, deliberate application** to achieve **safety, efficiency, and clarity** in modern C++ code.

Chapter 8

Text Encoding & Character Set Changes

8.1 UTF-8 as Source File Encoding

C++23 introduces **full support for UTF-8 as a portable source file encoding**. This change is significant for **modern software development**, especially in applications dealing with **internationalization, multilingual text, and Unicode-based systems**.

8.1.1 Motivation

Historically:

- C++ source files were encoded in **implementation-defined character sets**, often **ASCII, ANSI, or platform-specific encodings**.
- Non-ASCII characters in source code were **unreliable** across different compilers

and platforms.

- Unicode literals, identifiers, and string handling often required **workarounds or manual conversions**.

C++23 standardizes **UTF-8 encoding** as a **portable, predictable source file encoding**, simplifying **cross-platform development and internationalization**.

8.1.2 Key Changes

UTF-8 Source Files

- Source files can now be **saved and interpreted as UTF-8 by default**.
- Identifiers, string literals, and character literals in **UTF-8 are standardized**.
- Previous platform-dependent behaviors are removed, improving **portability and consistency**.

Example: UTF-8 source file:

```
#include <iostream>

int main() {
    std::string greeting = u8"Hello, !";
    std::cout << greeting << std::endl;
    return 0;
}
```

- `u8"Hello, !"` is a **UTF-8 string literal**.
- Source file **must be saved as UTF-8** for correct compilation.

Universal Character Names C++23 allows **named universal character escapes**:

```
std::cout << "\N{CAT FACE}" << std::endl; // prints
```

- Improves **readability** compared to numeric escapes (`\U0001F431`).
- Works consistently across **platforms and compilers**.

Delimited Escape Sequences

- New delimited escape sequences are standardized:

```
"\o{7777}"   // octal value
"\x{C0DE}"   // hexadecimal value
"\u{CAFE}"   // Unicode code point
```

- Makes **large code points readable and portable**.
- Reduces errors from **manual conversions**.

8.1.3 Implications for Modern C++

1. **Internationalization**
    - Developers can safely include **multilingual identifiers and strings**.
    - Reduces the need for **external encoding libraries**.
2. **Cross-Platform Consistency**

- UTF-8 encoded source files behave consistently across **Windows, Linux, macOS, and embedded systems**.

3. **Improved String Literals**

   - UTF-8 literals are compatible with **`std::string`** and **`std::u8string`**.
   - Avoids implicit conversions and **locale-dependent behaviors**.

4. **Better Compiler Diagnostics**

   - Compilers can **detect invalid UTF-8 sequences** at compile-time.
   - Reduces **undefined behavior or runtime encoding errors**.

8.1.4 Best Practices

- **Always save source files as UTF-8** when using non-ASCII characters.
- Prefer **named universal character escapes** for clarity in code.
- Use `u8""` literals for UTF-8 strings compatible with `std::string`.
- Avoid platform-specific encodings to maintain **portable code**.
- Combine UTF-8 encoding with **modern C++23 features** like `[[assume]]` and decay-copy for **robust, modern codebases**.

8.1.5 Summary

C++23's support for **UTF-8 as source file encoding** ensures:

- **Portability** across platforms and compilers.

- **Consistency** in string handling and identifiers.
- **Safe internationalization** without external libraries.
- **Readable and maintainable Unicode escapes**, both numeric and named.

This feature is essential for **modern C++ development**, especially in **multilingual, cross-platform, and high-quality software projects**.

8.2 Universal Character Escapes

C++23 introduces **enhanced support for universal character escapes**, making it easier to **represent Unicode characters** in source code reliably and portably. This improves **readability, portability, and internationalization**.

8.2.1 Motivation

Historically:

- Unicode characters could be represented with **`\u` and `\U` escape sequences**, e.g., `\u03A9` for Ω.
- Code using **multilingual characters** often required **manual numeric encoding**, making it **hard to read and error-prone**.
- Named Unicode characters were not standardized in source code, limiting **expressiveness and clarity**.

C++23 solves these problems by introducing:

1. **Named universal character escapes**
2. **Delimited escape sequences** for numeric code points

This allows **clear, maintainable, and portable source code** when working with Unicode.

8.2.2 Named Universal Character Escapes

```
"\N{CHARACTER NAME}"
```

Syntax

- `CHARACTER NAME` is the official Unicode character name.
- Produces a **UTF-8 character** in the source code.
- Works for **identifiers, string literals, and character literals**.

```
#include <iostream>
int main() {
    std::string cat = "\N{CAT FACE}";
    std::string heart = "\N{HEAVY BLACK HEART}";

    std::cout << cat << " " << heart << std::endl; // prints
    return 0;
}
```

Example: Named Escapes in Strings

- Improves **readability** compared to numeric escapes like `\U0001F431`.
- Eliminates **errors in code point conversion**.

```
int \N{GREEK SMALL LETTER ALPHA} = 5;
std::cout <<  << std::endl; // prints 5
```

Example: Named Escapes in Identifiers

- Allows **Unicode characters in identifiers** in a **portable and readable manner**.

8.2.3 Delimited Escape Sequences

C++23 introduces **clear and unambiguous escape sequences** for numeric code points:

```
"\o{7777}"   // octal escape
"\x{CODE}"   // hexadecimal escape
"\u{CAFE}"   // Unicode code point escape
```

- Delimiters `{}` avoid **ambiguity with adjacent characters**.
- Enables **large code points** to be represented safely.
- Improves **portability and compiler diagnostics**.

```
#include <iostream>
int main() {
    std::string smile = "\u{1F604}"; //
    std::cout << smile << std::endl;
    return 0;
}
```

Example: Delimited Escapes

- Clearer than traditional `\U0001F604`.
- Avoids mistakes in **multi-byte sequences**.

8.2.4 Integration with UTF-8 Source Files

- Named and delimited escapes **work naturally with UTF-8 encoded source files**.
- Combine with **`u8` string literals** for **consistent UTF-8 handling**:

```
std::string message = u8"Greetings: \N{WAVING HAND SIGN} \u{1F30D}";
std::cout << message << std::endl; //
```

- Ensures **cross-platform compatibility**.

8.2.5 Benefits of Universal Character Escapes

1. **Readability**
   - Named escapes are **self-documenting**.
   - Easier to maintain than numeric escapes.
2. **Portability**
   - Works consistently across **Windows, Linux, macOS, and embedded systems**.
3. **Error Reduction**
   - Avoids **manual conversion mistakes** for large Unicode code points.
4. **Internationalization & Multilingual Support**
   - Use **Unicode characters in strings, identifiers, and literals** safely.

5. **Tooling and Analysis Friendly**
    - Static analyzers and compilers can **recognize named characters** reliably.

8.2.6 Best Practices

- Prefer **named escapes** (`\N{CHARACTER NAME}`) when readability is important.
- Use **delimited numeric escapes** (`\u{}` or `\x{}`) for **large or uncommon code points**.
- Save source files as **UTF-8** for consistency.
- Avoid mixing **legacy encodings** with UTF-8 escapes.
- Combine with **modern C++23 features** like `[[assume]]`, lambda attributes, and decay-copy for **safe, expressive, and maintainable code**.

8.2.7 Summary

C++23's universal character escape support enables:

- **Named and readable Unicode escapes**
- **Delimitation for unambiguous numeric code points**
- **Portable, consistent, and error-free source code**
- **Enhanced internationalization support**
- Seamless integration with **UTF-8 source files**

This feature is essential for **modern, multilingual, and globally compatible C++ projects**.

8.3 Practical Unicode Examples

C++23 standardizes **UTF-8 source encoding and enhanced character escapes**, enabling developers to write **portable, readable, and internationalized code**. This section demonstrates **practical usage patterns** across strings, identifiers, and I/O.

8.3.1 Multilingual Strings

Modern applications often need **text in multiple languages**. C++23 allows **UTF-8 source files and named escapes** to simplify this:

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

int main() {
    std::string greeting = u8"English: Hello, World!";
    std::string japanese = u8"Japanese:     "; // UTF-8 source file
    std::string emoji = u8"Emoji: \N{GRINNING FACE} \N{EARTH GLOBE EUROPE-AFRICA}";

    std::cout << greeting << std::endl;
    std::cout << japanese << std::endl;
    std::cout << emoji << std::endl;

    return 0;
}
```

Highlights:

- `u8""` ensures the string is **UTF-8 encoded**.
- `\N{...}` named escapes increase **readability** of emoji and special symbols.
- Works reliably across **all modern compilers**.

8.3.2 Unicode in Identifiers

C++23 allows **Unicode characters in identifiers** if the source file is UTF-8 encoded:

```
#include <iostream>

int main() {
    int \N{GREEK SMALL LETTER ALPHA} = 10;
    int   = 20;
    std::cout <<   +   << std::endl; // prints 30
    return 0;
}
```

- Improves **semantic clarity** when using **domain-specific symbols**.
- Must ensure **all team members and tools** support UTF-8 source files.

8.3.3 Delimited Escape Sequences

C++23 introduces **clear numeric escapes** for large code points:

```
#include <iostream>

int main() {
    std::string smiley = "\u{1F604}";    //
    std::string coffee = "\u{2615}";     //
    std::string cat = "\N{CAT FACE}";    //

    std::cout << smiley << " " << coffee << " " << cat << std::endl;

    return 0;
}
```

Highlights:

- `\u{}` syntax avoids **ambiguity with adjacent characters**.
- Works for **emoji, symbols, and uncommon Unicode characters**.
- Fully compatible with **UTF-8 source files**.

8.3.4 Combining UTF-8 Literals and Standard Library

Using **modern C++23 string utilities** with UTF-8 strings:

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

int main() {
    std::string_view text = u8"  Hello,  !";
    std::cout << "Length in bytes: " << text.size() << std::endl;

    for (char c : text) {
        std::cout << std::hex << static_cast<int>(static_cast<unsigned char>(c)) << "
        ↪  ";
    }
    std::cout << std::endl;

    return 0;
}
```

- Iterating over **UTF-8 bytes** shows **raw encoding**.
- Can be combined with **std::ranges and views** for **modern string processing**.

8.3.5 Unicode in File I/O

C++23 ensures **consistent reading and writing of UTF-8 files**:

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

int main() {
    std::ofstream file("unicode.txt", std::ios::binary);
    file << u8"Hello,  !\n    \n";
    file.close();

    std::ifstream infile("unicode.txt", std::ios::binary);
    std::string line;
    while (std::getline(infile, line)) {
        std::cout << line << std::endl;
    }

    return 0;
}
```

- Ensures **cross-platform file portability**.
- UTF-8 encoding avoids **locale-dependent issues**.

8.3.6 Best Practices for Practical Unicode Usage

1. **Always save source files as UTF-8**.
2. Prefer **named escapes** for **readability** and clarity.
3. Use **delimited numeric escapes** for large code points.

4. Combine **UTF-8 literals (`u8`)** with **standard library strings**.
5. Test **I/O operations** across platforms to ensure **consistent behavior**.
6. Document **Unicode assumptions** for identifiers and literals.

8.3.7 Summary

C++23's **UTF-8 source files and universal character escapes** allow:

- Readable and maintainable **multilingual strings**.
- Safe inclusion of **Unicode characters in identifiers**.
- Portable **emoji, symbols, and special characters** in strings and I/O.
- Integration with **modern C++23 features** such as ranges and constexpr.

These capabilities are essential for **globalized, modern C++ applications** that need **robust, readable, and portable Unicode support**.

Chapter 9

`constexpr` Evolution

9.1 New Allowances (static/thread_local, gotos, non-literals)

C++23 introduces **important enhancements to `constexpr` functions** that extend **compile-time evaluation** beyond previous limitations. These changes improve **generic programming, metaprogramming, and safe compile-time computations**.

9.1.1 Motivation

Prior to C++23:

- `constexpr` functions were **restricted to literal types** and **compile-time evaluable operations**.
- **Static and thread-local variables** were not allowed.

- **Labels and goto statements** could not be used in `constexpr`.
- Non-literal variables or temporary objects could not participate in compile-time evaluation.

C++23 lifts these restrictions, allowing **more complex and realistic algorithms** to be computed at compile time, increasing **performance, safety, and expressiveness**.

9.1.2 Static and thread_local variables in `constexpr`

C++23 permits **static** and **thread_local** variables inside `constexpr` functions, with **well-defined semantics**:

```
#include <iostream>

constexpr int counter_increment() {
    static int counter = 0;   // static allowed in constexpr
    counter++;
    return counter;
}

int main() {
    constexpr int first = counter_increment();  // evaluated at compile time
    constexpr int second = counter_increment();
    std::cout << first << ", " << second << std::endl; // prints 1, 2
    return 0;
}
```

- **Static variables** maintain state across function calls.
- **thread_local variables** allow compile-time state per thread in multithreaded constexpr evaluation.

- Previously, **stateful compile-time functions** were impossible.

9.1.3 Non-literal types and objects

C++23 allows **constexpr functions to work with non-literal types**, significantly expanding **compile-time algorithm capabilities**:

```
#include <vector>
#include <array>

constexpr std::array<int, 3> create_array() {
    std::vector<int> temp{1, 2, 3}; // non-literal allowed
    std::array<int, 3> result{};
    for (size_t i = 0; i < temp.size(); ++i) {
        result[i] = temp[i];
    }
    return result;
}

constexpr auto arr = create_array();
```

- Non-literal types such as `std::vector` can now **participate in compile-time computations** indirectly.
- This allows **more realistic data structures** in constexpr context.

9.1.4 Labels and goto in `constexpr`

C++23 permits **labels and goto statements** in `constexpr` functions, enabling **flexible control flow**:

```cpp
#include <iostream>

constexpr int compute(int x) {
    int result = 0;
START:
    if (x > 0) {
        result += x;
        x--;
        goto START;
    }
    return result;
}

constexpr int sum = compute(5);  // sum = 15
```

- Makes `constexpr` **capable of loops with complex control flow**.
- Reduces reliance on **template metaprogramming for iteration**.

9.1.5 Benefits of These New Allowances

1. **More Expressive Compile-Time Functions**
    - Complex algorithms with **stateful logic, loops, and dynamic data** can now execute at compile time.
2. **Improved Generic Programming**
    - Allows **compile-time initialization of containers and data structures**.
3. **Better Performance**

- More computations **shifted to compile-time**, reducing runtime overhead.

4. **Simpler Metaprogramming**

   - Less reliance on **template recursion** or **SFINAE tricks** for complex compile-time logic.

9.1.6 Best Practices

- Use **static variables** for persistent compile-time state where necessary.
- Use **thread_local** only if thread-specific compile-time computations are required.
- Avoid **side effects** outside constexpr function context to ensure **predictable results**.
- Combine **non-literal support, static variables, and goto** to implement **compile-time algorithms previously impossible**.

9.1.7 Summary

C++23 extends `constexpr` capabilities by allowing:

- **Static and thread_local variables** inside `constexpr`.
- **Non-literal types** for compile-time evaluation.
- **Labels and goto statements** in `constexpr`.

These enhancements make `constexpr` **far more powerful and flexible**, bridging the gap between **runtime and compile-time computation**, and enabling **modern, high-performance, and maintainable C++23 code**.

9.2 How This Expands Compile-Time Computation

C++23 introduces **several enhancements to `constexpr`**, including **static/thread_local variables, non-literal types, labels, and goto statements**. These changes **dramatically expand the scope of computations possible at compile-time**, opening new opportunities for **high-performance, safe, and generic programming**.

9.2.1 Traditional Limitations

Prior to C++23:

- `constexpr` functions were limited to **literal types only**.
- **Static variables** could not retain state.
- **Thread-local storage** was forbidden in constexpr.
- **Labels and goto statements** were prohibited.
- Loops had to rely on **recursion or template metaprogramming** for compile-time iteration.

This restricted **compile-time computations** to **simple calculations** like arithmetic, basic container initialization, and literal constant evaluations.

9.2.2 Expanded Capabilities

Static and thread_local variables

- **Persistent state** is now allowed inside `constexpr`, making **stateful algorithms feasible at compile-time**.

- Example: cumulative counters, lookup tables, or memoization can now be **initialized and used entirely at compile-time**.

```
constexpr int fibonacci(int n) {
    static int memo[100] = {0}; // static array in constexpr
    if (n < 2) return n;
    if (memo[n] != 0) return memo[n];
    memo[n] = fibonacci(n-1) + fibonacci(n-2);
    return memo[n];
}

constexpr int fib10 = fibonacci(10);  // fib10 = 55
```

- This previously required **template metaprogramming or recursive constexpr hacks**.

Non-literal types

- Containers like `std::vector` or `std::string` can now **participate indirectly** in compile-time computations.
- Enables **realistic data structures** for compile-time initialization and manipulation.

```
#include <array>
#include <vector>

constexpr std::array<int, 3> generate_array() {
    std::vector<int> tmp{10, 20, 30};
    std::array<int, 3> arr{};
```

```
    for (size_t i = 0; i < tmp.size(); ++i) arr[i] = tmp[i];
    return arr;
}

constexpr auto arr = generate_array();
```

- Complex **algorithms, transformations, and preprocessing** can now happen **at compile-time**.

Labels and goto

- Control flow flexibility is significantly improved.
- Allows **loops and branches** inside constexpr without recursion or templates.

```
constexpr int sum_to(int n) {
    int result = 0;
    START:
        if (n <= 0) return result;
        result += n;
        n--;
        goto START;
}
constexpr int total = sum_to(10); // total = 55
```

- Makes **iterative compile-time computation** natural and readable.

9.2.3 Combined Impact

C++23 allows **arbitrary algorithms** at compile-time that were previously **limited to runtime**:

- **Memoization of expensive computations**
- **Compile-time data structure initialization**
- **Static configuration tables** for embedded and high-performance applications
- **Generic programming** where type traits and constants are computed fully at compile-time

This **reduces runtime overhead**, enabling **more efficient, safer, and predictable programs**.

9.2.4 Integration with Other Features

- Works seamlessly with **`if consteval` / `if not consteval`**, allowing **conditional execution at compile-time vs runtime**.
- Can be combined with **C++23 ranges, constexpr algorithms, and non-literal container operations**.

Example combining features:

```
#include <array>
#include <ranges>
constexpr auto create_lookup_table() {
    std::array<int, 5> table{};
    for (int i = 0; i < 5; ++i) {
        table[i] = i * i; // compile-time calculation
    }
    return table;
}

constexpr auto squares = create_lookup_table();
```

- The table `squares` is fully initialized **at compile-time**, ready for **high-performance use in runtime code**.

9.2.5 Benefits

1. **Performance** – Move complex computations to compile-time, reducing runtime cost.
2. **Safety** – Compile-time guarantees prevent errors from propagating to runtime.
3. **Expressiveness** – Stateful, iterative, and complex logic is now feasible in `constexpr`.
4. **Generic Programming** – Supports more flexible template and metaprogramming patterns.
5. **Maintainability** – Readable, iterative compile-time algorithms replace convoluted template hacks.

9.2.6 Best Practices

- Use **static variables** for persistent compile-time state only when necessary.
- Prefer **non-literal container operations** for **realistic compile-time data processing**.
- Combine **labels and goto** carefully for **clear and maintainable control flow**.
- Test **compile-time behavior** across multiple compilers to ensure compliance.
- Integrate **with ranges, constexpr algorithms, and conditional constexpr logic** for maximum efficiency.

9.2.7 Summary

C++23's expanded `constexpr` capabilities:

- Introduce **stateful compile-time computation** using static/thread_local variables.
- Allow **non-literal types** and **complex data structures** in constexpr.
- Enable **labels and goto statements**, providing flexible control flow.
- Significantly **increase the scope and power of compile-time computation**, bridging runtime and compile-time logic.

These enhancements **empower developers** to write **high-performance, maintainable, and expressive modern C++23 code**.

Chapter 10

Other Language Tweaks

10.1 CTAD from Inherited Constructors

Class Template Argument Deduction (CTAD) was introduced in **C++17** to **deduce template parameters from constructor arguments**, making class template usage simpler and more intuitive. C++23 expands CTAD to **support inherited constructors**, enabling derived classes to **automatically deduce template arguments from base constructors**.

10.1.1 Motivation

Before C++23:

- CTAD worked only for constructors **directly declared in the class template**.
- If a derived class inherited a base class constructor, **CTAD did not apply**, forcing developers to **explicitly specify template arguments**.

C++23 solves this problem by **allowing inherited constructors to participate in CTAD**, making **template-based class hierarchies more concise and easier to maintain**.

10.1.2 Syntax

CTAD from inherited constructors is enabled automatically by **declaring `using Base::Base;`** in a derived template class:

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

template <typename T>
class Base {
public:
    Base(T value) { std::cout << "Base constructor: " << value << "\n"; }
};

template <typename T>
class Derived : public Base<T> {
public:
    using Base<T>::Base; // inherit constructors
};

int main() {
    Derived d1(42);       // CTAD deduces T = int
    Derived d2(std::string("Hello")); // CTAD deduces T = std::string
    return 0;
}
```

Highlights:

- The `using Base<T>::Base;` syntax **inherits constructors** from the base template.
- C++23 **deduces `T` automatically** from constructor arguments.
- No explicit template parameter is needed, making **class instantiation simpler and more readable**.

10.1.3 Practical Benefits

1. **Reduced Boilerplate**

```
Derived<int> d1(10);  // pre-C++23 required explicit template argument
Derived d2(20);        // C++23 allows deduction directly
```

- Eliminates repetitive template argument specification when inheriting constructors.

1. **Improved Generic Programming**

- CTAD from inherited constructors allows **more flexible class hierarchies** with **templates**, improving **code reusability**.

1. **Readable and Maintainable Code**

- Developers can **focus on class design and logic**, rather than explicit template syntax.

1. **Seamless Integration**

- Works with **other C++23 features** like `constexpr` constructors, ranges, and `std::expected` or `std::optional` wrappers.

10.1.4 Edge Cases & Considerations

- If a derived class has its own constructors **that conflict** with inherited ones, CTAD might deduce **unexpected types**.
- Multiple base classes with **overlapping constructor signatures** require careful management to avoid **ambiguity**.
- CTAD from inherited constructors applies only if **template parameters can be deduced** from arguments.

10.1.5 Examples in Real-World Programming

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

template <typename T>
class MyVector : public std::vector<T> {
public:
    using std::vector<T>::vector;
};

int main() {
    MyVector<int> mv1{1,2,3};          // deduces T = int
    MyVector<std::string> mv2{"a","b"}; // deduces T = std::string
}
```

Container Wrappers

- Wraps standard containers while **preserving CTAD convenience**.

```cpp
#include <memory>
#include <string>

template <typename T>
class Resource {
public:
    Resource(T val) : data(std::make_shared<T>(val)) {}
    std::shared_ptr<T> data;
};

template <typename T>
class ManagedResource : public Resource<T> {
public:
    using Resource<T>::Resource;
};

int main() {
    ManagedResource<int> r1(100); // CTAD deduces T=int
    ManagedResource<std::string> r2("Hello"); // T=std::string
}
```

Custom Resource Management

- Simplifies creation of **resource-managing template classes**.

10.1.6 Summary

C++23 allows **Class Template Argument Deduction from inherited constructors**, providing:

- Automatic deduction of template parameters from base class constructors.

- Cleaner, more maintainable class hierarchies.
- Reduced boilerplate and easier generic programming.
- Seamless integration with other modern C++ features.

This enhancement **bridges the gap between inheritance and template deduction**, making C++23 **more expressive, concise, and developer-friendly**.

10.2 Range-for Loop Lifetime Extension

C++23 introduces a **refined rule for extending the lifetime of temporaries created in range-based for loops**, ensuring **that temporaries used as the loop range remain valid for the loop body**.

10.2.1 Motivation

Previously:

- In **C++11–C++20**, range-based for loops could bind **temporaries in ways that were subtle and error-prone**.
- Example:

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

std::vector<int> generate_vector() {
    return {1, 2, 3};
}

int main() {
    for (int x : generate_vector()) {  // temporary returned by generate_vector
        std::cout << x << "\n";       // temporary must survive the loop
    }
}
```

- Although in most implementations this worked due to compiler extensions, **the temporary's lifetime was not formally guaranteed by the standard**.

- Developers had to rely on **careful coding patterns** to avoid dangling references.

C++23 makes this **formally safe and standardized**, ensuring **all temporaries used as range expressions live through the loop body**.

10.2.2 How Lifetime Extension Works

In C++23:

- The **temporary object** created for the range expression is **automatically extended** to cover the **entire loop body**.
- Both **value and reference-based loops** benefit from this guarantee.

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

struct Wrapper {
    int value;
};

std::vector<Wrapper> make_wrappers() {
    return {{1}, {2}, {3}};
}

int main() {
    for (auto& w : make_wrappers()) {  // reference binds to temporary safely
        std::cout << w.value << "\n";
    }
}
```

- The temporary returned by `make_wrappers()` is **kept alive for the duration of the loop**, preventing **dangling references**.

10.2.3 Examples

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

for (auto x : std::vector<int>{10, 20, 30}) {
    std::cout << x << "\n"; // Safe in C++23; temporary vector survives
}
```

Simple Range-For with Temporary

- Before C++23, using a temporary directly like this could be **implementation-defined behavior**.

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

struct Data { int val; };

for (auto& d : std::vector<Data>{{1}, {2}, {3}}) {
    d.val += 10;   // Safe; temporary survives loop
    std::cout << d.val << "\n";
}
```

Reference Loop

- Modifying references to temporaries is now **well-defined**.

```
for (auto&& elem : std::vector<int>{5, 6, 7}) {
    elem *= 2;  // Safe; works for rvalue and lvalue ranges
}
```

Integration with `auto&&`

- Enables **perfect forwarding in range-based loops** without lifetime issues.

10.2.4 Benefits

1. **Improved Safety**

- Prevents **dangling references** when iterating over temporary objects.

1. **Simpler and Cleaner Code**

- Developers no longer need **workarounds** like storing the temporary in a local variable before looping.

1. **Better Generic Programming**

- Templates and generic algorithms that use **range-based for loops over temporaries** are now **fully standard-compliant**.

1. **Enhanced Expressiveness**

- Enables concise constructs like:

```
for (auto x : compute_values()) {
    // use x directly
}
```

- Where `compute_values()` returns a temporary container.

10.2.5 Edge Cases & Considerations

- Lifetime extension applies **only to the range expression itself**, not to **temporaries created inside the loop body**.
- References to **elements of the temporary container** remain valid throughout the loop.
- Compatible with **structured bindings**:

```
for (auto [key, value] : std::vector<std::pair<int,int>>{{1,2},{3,4}}) {
    std::cout << key << ":" << value << "\n";
}
```

- Lifetime of the vector temporary is **guaranteed for the loop body**.

10.2.6 Best Practices

- Prefer **direct range-for loops over temporaries** for concise code.
- Use **`auto&` or `auto&&`** for reference iteration to avoid unnecessary copies.
- Leverage this feature in **generic templates**, especially when working with **computed or temporary containers**.

- Ensure **nested temporaries** inside the loop are handled explicitly if needed; only the outermost range temporary gets automatic lifetime extension.

10.2.7 Summary

C++23 formalizes **lifetime extension for temporaries in range-based for loops**, providing:

- **Safe and predictable behavior** for loops over temporaries.
- **Cleaner syntax** without extra local variables.
- **Seamless integration with references, auto&&, and structured bindings**.
- **Improved generic programming and maintainable code**.

This enhancement may seem subtle, but it **eliminates a class of potential bugs**, making **modern C++23 loops safer and more expressive**.

10.3 Optional Parentheses in Lambdas

C++23 introduces the **ability to omit the empty parentheses `()` in lambda expressions** when the lambda takes **no parameters**, making **inline functions more concise**.

10.3.1 Motivation

Previously (C++11–C++20):

- Lambdas with no parameters **required parentheses**:

```
auto f = []() { return 42; };
std::cout << f() << "\n"; // prints 42
```

- While functional, this syntax was **verbosely repetitive**, especially in **generic programming and functional-style code**.
- C++23 **removes this requirement**, enabling **cleaner, more expressive code**.

10.3.2 Syntax

C++23 allows **empty-parameter lambdas** to be written **without parentheses**:

```
auto f = [] { return 42; }; // parentheses omitted
std::cout << f() << "\n";   // prints 42
```

Key rules:

- Parentheses can be omitted **only if the lambda takes zero parameters**.

- Capture lists (`[]`) remain **mandatory**.
- Body syntax `{ ... }` is unchanged.

10.3.3 Examples

```
#include <iostream>

int main() {
    auto greet = [] { std::cout << "Hello, C++23!\n"; };
    greet(); // prints "Hello, C++23!"
}
```

Simple Lambda

- This is **equivalent to `[]() {...}`**, but cleaner.

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

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    std::for_each(v.begin(), v.end(), [](int x){ std::cout << x*x << " "; }); // old
    ↪ style
    std::for_each(v.begin(), v.end(), [=]{ std::cout << "done\n"; }); // C++23 style,
    ↪ parentheses omitted
}
```

In STL Algorithms

- Works well for **parameterless callbacks**, such as final actions or notifications.

```
auto outer = [] {
    auto inner = [] { return 100; }; // parentheses omitted
    return inner();
};

std::cout << outer() << "\n"; // prints 100
```

Nested Lambdas

- Improves **nested lambda readability** in functional code.

10.3.4 Benefits

1. **Conciseness**

- Reduces boilerplate for **parameterless lambdas**.

1. **Improved Readability**

- Code appears **less cluttered**, particularly in **algorithm pipelines** and **callbacks**.

1. **Consistent Style**

- Matches **other modern C++ features** like structured bindings, `auto` deduction, and range-based for loops.

1. **Better Generic and Functional Programming**

- Simplifies **lambda-based functional constructs**, especially when used in **template code**.

10.3.5 Edge Cases & Considerations

- **Parentheses cannot be omitted** if the lambda has any parameters:

```
auto f = [](int x) { return x*2; }; // parentheses required
```

- Capture-only lambdas with no parameters can fully omit `()`:

```
int y = 5;
auto f = [y] { return y*2; }; // OK
```

- Works seamlessly with **constexpr lambdas**, **auto parameters**, and **generic lambdas** with zero parameters.

10.3.6 Best Practices

- Use **optional parentheses** for **parameterless lambdas**, especially in **inline callbacks**.
- Maintain parentheses for **clarity** in complex or nested lambdas if it improves readability.
- Combine with **range-based for loops**, **STL algorithms**, and **functional-style code** for **concise modern C++23 patterns**.

10.3.7 Summary

C++23 allows **omitting parentheses for lambdas with no parameters**, providing:

- **Cleaner, more concise lambda syntax**.
- **Improved readability** in functional and generic programming.
- Seamless integration with **modern C++ patterns**, **range-based loops**, and **STL algorithms**.

This small tweak enhances **expressiveness** and **reduces syntactic noise**, aligning C++ with **developer expectations for modern, readable code**.

10.4 Labels at the End of Compound Statements

This section provides a **detailed exploration of the enhanced control flow flexibility in C++23**, specifically **allowing labels at the end of compound statements**, a subtle but useful improvement for **goto and structured programming**.

10.5 Labels at the End of Compound Statements

C++23 introduces the ability to **place labels at the end of a compound statement** (i.e., a block enclosed in `{ ... }`). This small syntactic tweak enables **more flexible code organization**, particularly when using **goto statements for early exits, error handling, or cleanup routines**.

10.5.1 Motivation

Previously (C++98–C++20):

- Labels could only appear **before a statement**, not at the end of a block:

```
void example(int x) {
    if (x > 0) {
        std::cout << "Positive\n";
        // END:  //  Not allowed in C++20
    }
}
```

- To perform **early jumps or structured cleanup**, developers had to **reorder code** or **add extra statements**, which could make code **less readable**.

C++23 **relaxes this restriction**, allowing labels at the **end of compound statements**, improving **readability and maintainability**, especially in **manual resource management or low-level code**.

10.5.2 Syntax

The new syntax allows **labels at the end of a block**:

```
void process(int& x) {
    if (x != 0) {
        x *= 2;
    END:
        ; // optional statement
}
```

- The label `END:` is now **valid** at the end of the compound statement.
- The block can optionally include a **null statement (`;`)** after the label.

Key points:

- Works in **all scopes** where labels are legal.
- Enables **goto jumps from outside the block** into the label.
- Fully compatible with **nested blocks, loops, and conditionals**.

10.5.3 Examples

```
#include <iostream>

void compute(int& x) {
    if (x < 0) goto END; // jump to label at end
    x *= 10;
END:
```

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

int main() {
    int a = 5;
    int b = -3;
    compute(a); // prints 50
    compute(b); // prints -3
}
```

Simple Early Exit

- Label placement at the **end of block** allows **early exit without additional statements**.

```
for (int i = 0; i < 5; ++i) {
    for (int j = 0; j < 5; ++j) {
        if (i * j > 6) goto OUTER_END;
    }
}
OUTER_END:
std::cout << "Exited loops early\n";
```

Nested Loops

- Simplifies **multi-level jumps**, common in **low-level algorithms or resource cleanup routines**.

```
{
    int x = 42;
    if (x > 0) {
        std::cout << "Positive\n";
    }
LABEL_END:
    ; // label at end
}
```

Integration with Structured Blocks

- Enables **flexible code labeling** without rearranging statements.

10.5.4 Benefits

1. **Improved Flexibility**

- Labels can now **exist anywhere in blocks**, not just before statements.
- Useful for **goto-based error handling** or **cleanup patterns**.

1. **Cleaner Control Flow**

- Reduces the need for **dummy statements** or **restructured code**.
- Makes **manual control flow** more intuitive.

1. **Better Low-Level Programming Support**

- Useful for **embedded systems**, **OS kernels**, or **high-performance code** where **goto** may still be used responsibly.

1. **Backward Compatibility**

- Code using labels in traditional positions still works.
- No impact on **existing C++ syntax**, only adds **new allowed positions**.

10.5.5 Edge Cases & Considerations

- Using **goto to jump into a block** remains restricted by **variable initialization rules**: cannot jump past **variables with non-trivial constructors**.
- Labels at the end of blocks **cannot bypass initialization** of local variables.
- Best practice: use **labels at the end for readability**, not as a substitute for **structured programming** where exceptions or functions are preferred.

10.5.6 Summary

C++23 introduces **labels at the end of compound statements**, providing:

- **Enhanced control flow flexibility** for goto-based patterns.
- **Cleaner and more readable early-exit blocks**.
- **Safe integration with nested blocks and loops**.
- A minor but useful improvement for **low-level, embedded, and high-performance code**.

This feature, while small, aligns with C++23's philosophy of **expressiveness, safety, and minimal syntactic friction**.

10.6 Narrowing Boolean Conversion in `static_assert`

C++23 introduces a **refined rule for boolean conversions** in `static_assert` expressions, ensuring that **only values that are safely convertible to `bool` are allowed**. This **prevents unintended behavior caused by implicit narrowing conversions**, improving **compile-time safety and expressiveness**.

10.6.1 Motivation

Previously (C++11–C++20):

- `static_assert` accepted any expression that could be **contextually converted to `bool`**, even if that conversion was **implicit and potentially lossy**:

```
static_assert(42, "This will compile in C++20"); // 42 implicitly converts to true
```

- While technically correct, this could **mask programmer mistakes** and **reduce the clarity of compile-time assertions**.

C++23 **enforces narrowing rules**, making `static_assert` **fail if the expression is not explicitly safe for boolean conversion**. This change promotes **clear intent and safer metaprogramming**.

10.6.2 Syntax & Behavior

```
static_assert(1, "Allowed");  // OK, 1 converts to true
static_assert(0, "Allowed");  // OK, 0 converts to false
static_assert(42, "Allowed"); // OK, implicitly converts to true (but potentially
↪  unintended)
```

Before C++23

- Implicit numeric-to-boolean conversions were allowed without warnings.

```
static_assert(true, "Allowed");   // OK
static_assert(false, "Allowed");  // OK
static_assert(42, "Error");       //  Narrowing conversion not allowed
```

In C++23
Rules:

1. Expressions in `static_assert` must **directly evaluate to `bool`** or be **explicitly convertible to `bool`**.
2. Implicit narrowing conversions (like integers other than `0` or `1`) are now **diagnosed as errors**.
3. Promotes **intentional and safe compile-time checks**.

10.6.3 Examples

```
static_assert(sizeof(int) == 4, "Integers must be 4 bytes"); // OK
static_assert(true, "Always true");                          // OK

constexpr bool is_valid(int x) { return x > 0; }
static_assert(is_valid(10), "x must be positive");           // OK
```

Correct Usage

```
static_assert(42, "Narrowing conversion"); //  Error in C++23
static_assert(3.14, "Narrowing conversion"); //  Error in C++23
```

Incorrect / Narrowing Conversion

- C++23 requires **explicit conversion** if you want to assert on non-boolean values:

```
static_assert(static_cast<bool>(42), "Explicit conversion OK"); //
```

10.6.4 Benefits

1. **Improved Compile-Time Safety**

- Prevents **accidental misuse** of `static_assert` with numeric or pointer values that might compile but not match intent.

1. **Clearer Code Semantics**

- Ensures the **assertion logic clearly evaluates to true or false**, improving readability and maintainability.

1. **Better Metaprogramming**

- Templates and constant expressions rely heavily on `static_assert`. **Tighter rules** reduce subtle bugs in **generic code**.

1. **Alignment with `if constexpr`**

- The same narrowing rules apply to **`if constexpr` expressions**, making **compile-time control flow predictable and safe**.

10.6.5 Edge Cases & Considerations

- **Pointers**: null vs. non-null conversions are allowed only if explicit or clearly boolean.

```
int* ptr = nullptr;
static_assert(ptr == nullptr, "Pointer must be null"); // OK
```

- **Explicit casting** is the recommended way for numeric types when necessary.
- Helps **diagnose legacy code** that relied on implicit integer-to-bool conversions in compile-time checks.

10.6.6 Summary

C++23 introduces **narrowing rules for `static_assert` and `if constexpr`**:

- Prevents **unsafe or unintended implicit conversions to `bool`**.
- Promotes **clear, intentional, and maintainable compile-time assertions**.
- Enhances **metaprogramming safety** and **compile-time code reliability**.
- Aligns C++ with modern practices of **explicit conversions and semantic clarity**.

This seemingly small tweak ensures that **compile-time checks are robust, intentional, and expressive**, consistent with C++23's philosophy of **safety, efficiency, and expressiveness**.

Part III

The C++23 Standard Library

Chapter 11

New Modules: `<std>` and `<std.compat>`

11.1 Introduction to Modules in C++23

C++23 introduces a significant advancement in the language's modularity with the standardization of two primary library modules: `std` and `std.compat`. These modules aim to enhance the efficiency and organization of C++ codebases by providing a more structured approach to importing the standard library.

11.2 The `std` Module

The `std` module serves as the core module for the C++ standard library. By importing `std`, developers gain access to all declarations within the `std` namespace, encompassing the functionalities provided by the standard C++ headers. This includes components such as algorithms, containers, iterators, and other utilities that are integral to C++

programming.

Key Features:

- **Comprehensive Access:** Provides access to all standard C++ library components within the `std` namespace.
- **Enhanced Compilation:** Facilitates faster compilation times by reducing the overhead associated with traditional header inclusion.
- **Improved Code Organization:** Encourages better code structure and modularity, leading to more maintainable codebases.

Usage Example:

```
import std;

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    // Further code utilizing standard library components
}
```

11.3 The `std.compat` Module

The `std.compat` module extends the functionalities of the `std` module by including additional declarations that are provided by the C++ headers for C library facilities. This module is particularly beneficial for projects that require compatibility with C standard library functions.

Key Features:

- **C Library Compatibility:** Includes declarations from C library headers, facilitating interoperability between C and C++ code.

- **Global Namespace Declarations:** Exports certain functions in the global namespace, aligning with the C standard library's structure.

- **Seamless Integration:** Ensures that C library functions are readily available in C++ projects without the need for separate header inclusions.

Usage Example:

```
import std.compat;

int main() {
    std::printf("Hello, World!\n");
    // Further code utilizing C library functions
}
```

11.4 Comparison of `std` and `std.compat`

Feature	`std` Module	`std.compat` Module
Namespace Access	`std` namespace	`std` namespace and global namespace
C Library Functions	Not included	Includes C library functions
Use Case	Standard C++ functionalities	C and C++ interoperability

11.5 Compiler Support and Implementation

As of C++23, the support for these modules is still in the early stages, with varying levels of implementation across different compilers. For instance, Clang and GCC have begun to incorporate support for these modules, but developers may need to enable specific flags or configurations to utilize them effectively.

Clang Example:

```
clang++ -std=c++23 -fmodules -o my_program my_program.cpp
```

GCC Example:

```
g++ -std=c++23 -fmodules-ts -o my_program my_program.cpp
```

It's important to consult the documentation of the respective compiler to understand the exact steps and configurations required to enable module support.

11.6 Benefits of Using Modules

- **Faster Compilation:** Modules can significantly reduce compilation times by eliminating redundant parsing of header files.
- **Better Encapsulation:** They provide a clearer separation between interface and implementation, leading to more maintainable code.
- **Improved Error Checking:** Modules facilitate more precise error diagnostics, aiding developers in identifying issues more efficiently.Stack Overflow

11.7 Conclusion

The introduction of the `std` and `std.compat` modules in C++23 marks a pivotal step towards modernizing the C++ language, aligning it with contemporary programming practices. While the adoption of these modules is still in progress, their potential benefits in terms of compilation efficiency, code organization, and interoperability are substantial. As compiler support becomes more robust, these modules are expected to play a central role in the development of future C++ applications.

Chapter 12

New Headers Overview

12.1 <expected>, <flat_map>, <flat_set>, <generator>, <mdspan>, <print>, <stacktrace>, <spanstream>, <stdfloat>

The C++23 standard introduces a broad set of new headers that enrich the functionality of the Standard Library. These headers address long-standing gaps in error handling, data representation, coroutines, I/O, diagnostics, and numerical computing. Each one embodies C++23's design goals: **safety, performance, and expressiveness**.

12.1.1 <expected>

The <expected> header introduces std::expected<T, E>, a new vocabulary type for error handling.

- **Purpose:** Provides a type-safe alternative to exceptions or raw error codes.

- **Structure:**
  - Holds either a value of type `T` (success case) or an error of type `E` (failure case).
  - Includes `std::unexpected<E>` and `std::bad_expected_access<E>` for handling error states.
- **Monadic Operations:** Offers functional combinators (`and_then`, `transform`, `or_else`) enabling error-aware chaining of operations.
- **Benefit:** Makes error handling explicit, composable, and efficient—particularly useful in environments where exceptions are discouraged.

Example:

```
#include <expected>
#include <string>

std::expected<int, std::string> parse_number(const std::string& s) {
    try {
        return std::stoi(s);
    } catch (...) {
        return std::unexpected("Invalid number");
    }
}

int main() {
    auto result = parse_number("42");
    if (result) {
        // Success path
    } else {
        // Error path: result.error()
```

```
    }
}
```

12.1.2 <flat_map> and <flat_set>

These headers define **flat associative containers**: `std::flat_map`, `std::flat_multimap`, `std::flat_set`, and `std::flat_multiset`.

- **Implementation:** Typically backed by a sorted `std::vector`, not a tree.
- **Characteristics:**
  - Cache-friendly due to contiguous storage.
  - Very fast lookups when data is stable.
  - Insertions and deletions are more costly (require shifting).
- **Use Cases:** Suitable for scenarios with heavy lookup workloads and relatively infrequent modifications.

Example:

```
#include <flat_map>
#include <string>

int main() {
    std::flat_map<std::string, int> scores = {{"Alice", 90}, {"Bob", 85}};
    scores["Charlie"] = 95;  // maintains sorted order internally
}
```

12.1.3 <generator>

This header introduces `std::generator`, a coroutine-based type for producing sequences lazily.

- **Purpose:** Provides a standard mechanism for writing generators without external libraries.
- **Integration:** Works seamlessly with C++ coroutines (`co_yield`).
- **Benefit:** Produces elements on demand, avoiding full materialization of sequences.

Example:

```
#include <generator>
#include <iostream>

std::generator<int> count_to(int n) {
    for (int i = 1; i <= n; ++i)
        co_yield i;
}

int main() {
    for (int x : count_to(5))
        std::cout << x << " ";
}
```

12.1.4 <mdspan>

The `<mdspan>` header standardizes `std::mdspan`, a **multi-dimensional, non-owning view** over contiguous data.

- **Analogy:** Similar to `std::span`, but for N-dimensional arrays.
- **Components:**
  - `std::extents` for dimension sizes.
  - Layout policies (`layout_left`, `layout_right`, `layout_stride`).
  - Accessor policies for indexing.
- **Benefit:** Offers performance portability and flexibility for high-performance computing and numerical applications.

Example:

```
#include <mdspan>
#include <vector>

int main() {
    std::vector<double> data(6);
    std::mdspan<double, std::extents<size_t, 2, 3>> matrix(data.data());
    matrix(1, 2) = 42.0;  // assign element at row 1, col 2
}
```

12.1.5 <print>

This header adds high-level printing functions such as `std::print` and `std::println`.

- **Purpose:** Simplifies output compared to `std::cout` or `printf`.
- **Features:**
  - Type-safe formatting.

- Direct output to standard streams or files.
- Format syntax aligned with `<format>` introduced in C++20.

Example:

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

int main() {
    std::string name = "World";
    std::println("Hello, {}", name);
}
```

12.1.6 `<stacktrace>`

This diagnostic library introduces `std::stacktrace` for capturing execution call stacks at runtime.

- **Key Types:**
  - `std::stacktrace_entry`: Represents a single frame.
  - `std::stacktrace`: Collection of frames.
- **Use Cases:** Error diagnostics, crash reporting, and debugging.

Example:

```
#include <stacktrace>
#include <iostream>
```

```
void foo() {
    std::cout << std::stacktrace::current();
}

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

12.1.7 <spanstream>

Provides span-based I/O stream classes (`std::spanstream`, `std::ispanstream`, `std::ospanstream`).

- **Purpose:** Like `std::stringstream`, but works over a fixed buffer defined by `std::span<char>`.
- **Benefit:** Avoids dynamic allocations, improves efficiency in constrained environments.

Example:

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

int main() {
    std::array<char, 64> buffer{};
    std::ospanstream out(buffer);
    out << "Pi is approximately " << 3.14159;

    std::ispanstream in(buffer);
```

```
    std::string text;
    std::getline(in, text);
    std::cout << text;
}
```

12.1.8 <stdfloat>

This header conditionally introduces **extended floating-point types** depending on platform support:

- `std::float16_t`
- `std::bfloat16_t`
- `std::float32_t`
- `std::float64_t`
- `std::float128_t`

Purpose: Provides standardized access to hardware-specific floating-point types often used in graphics, machine learning, and scientific computing.

- **Benefits:**
  - Portability of specialized numeric code.
  - Precision selection depending on performance or storage requirements.

Example:

```cpp
#include <stdfloat>

int main() {
#ifdef __STDCPP_FLOAT16_T__
    std::float16_t half_precision = 1.0f16;
#endif
}
```

12.1.9 Summary

The new headers in C++23 significantly extend the breadth of the standard library:

- **<expected>** — safer error/result handling.
- **<flat_map>, <flat_set>** — cache-friendly associative containers.
- **<generator>** — coroutine-driven lazy sequences.
- **<mdspan>** — multi-dimensional views for high-performance computing.
- **<print>** — simplified, modern formatted output.
- **<stacktrace>** — runtime diagnostics with call stack capture.
- **<spanstream>** — efficient fixed-buffer I/O streams.
- **<stdfloat>** — extended floating-point types for specialized workloads.

Together, they modernize C++ by improving **expressiveness, safety, performance, and debugging capabilities**, addressing both long-standing community requests and emerging application needs.

Chapter 13

General Utilities

13.1 `std::expected`, `std::move_only_function`, `std::bind_back`, `std::byteswap`, `std::forward_like`, `std::invoke_r`, `std::to_underlying`, `std::unreachable`

The C++23 standard library introduces a number of new general-purpose utilities that aim to simplify error handling, function wrapping, invocation, and low-level operations. These utilities extend the expressive power of C++ while adhering to the principles of type safety, efficiency, and clarity.

13.1.1 `std::expected`

The utility `std::expected<T, E>` represents either a value of type `T` (the expected result) or an error of type `E`.

- **Motivation:** Provides a type-safe alternative to exceptions, allowing developers to handle errors explicitly.
- **Structure:**
  - Holds one of two states: a `value()` or an `error()`.
  - Includes `std::unexpected<E>` and `std::bad_expected_access<E>` to support error handling.
- **Monadic Operations:** Functions like `transform`, `and_then`, `or_else` allow functional composition and chaining.
- **Use Case:** Ideal for APIs where error cases must be considered at the call site without relying on exception handling.

Example:

```
#include <expected>
#include <string>

std::expected<int, std::string> parse(const std::string& s) {
    try {
        return std::stoi(s);
    } catch (...) {
        return std::unexpected("Invalid integer");
    }
}

int main() {
    auto result = parse("42");
    if (result) {
        int value = result.value();
```

```
    } else {
        auto err = result.error();
    }
}
```

13.1.2 `std::move_only_function`

`std::move_only_function` is a general-purpose polymorphic function wrapper similar to `std::function`, but move-only instead of copyable.

- **Motivation:** `std::function` requires copyability of the wrapped callable, which can be restrictive.
- **Features:**
  - Can wrap callables that are move-only (like `std::unique_ptr`-based functors or lambdas with move-only captures).
  - Supports type-erased invocation with the same syntax as `std::function`.
- **Use Case:** Useful when designing APIs that should accept callables which cannot or should not be copied.

Example:

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

int main() {
    auto ptr = std::make_unique<int>(42);
```

```
    std::move_only_function<void()> func = [p = std::move(ptr)] {
        std::cout << *p << '\n';
    };
    func();
}
```

13.1.3 `std::bind_back`

This utility allows binding of arguments to the **end** of a callable's parameter list, as opposed to `std::bind_front` introduced in C++20.

- **Purpose:** Provides a symmetrical counterpart to `std::bind_front`.
- **Example Use Case:** Partial application of functions where trailing arguments are commonly fixed.

Example:

```
#include <functional>
#include <iostream>

void print_sum(int a, int b, int c) {
    std::cout << (a + b + c) << '\n';
}

int main() {
    auto f = std::bind_back(print_sum, 10, 20);
    f(5); // Equivalent to print_sum(5, 10, 20)
}
```

13.1.4 `std::byteswap`

Introduced as a standardized way to reverse the byte order of integral types.

- **Motivation:** Endianness conversion is a common requirement in networking, file formats, and low-level systems programming.
- **Features:**
  - Accepts any integral type.
  - Returns a value of the same type with its bytes reversed.

Example:

```
#include <bit>
#include <cstdint>
#include <iostream>

int main() {
    uint32_t x = 0x12345678;
    auto swapped = std::byteswap(x); // 0x78563412
    std::cout << std::hex << swapped << '\n';
}
```

13.1.5 `std::forward_like`

This utility enables forwarding of value category (lvalue/rvalue) **like another object**.

- **Motivation:** Often when forwarding parameters, you want the forwarding behavior to match that of another reference, not just the local one.

- **Use Case:** Forwarding arguments in templated functions when maintaining consistency with another parameter's value category.

Example:

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

void foo(int& x) { std::cout << "lvalue\n"; }
void foo(int&& x) { std::cout << "rvalue\n"; }

template <typename T, typename U>
void call_like(U&& u, T&& t) {
    foo(std::forward_like<U>(t));
}

int main() {
    int a = 10;
    call_like(a, 20);    // forwards as lvalue
    call_like(42, 20);   // forwards as rvalue
}
```

13.1.6 `std::invoke_r`

This is an extension of `std::invoke` introduced earlier.

- **Purpose:** Invokes a callable, but explicitly converts the result to a given return type.
- **Benefit:** Provides explicit control over return types, improving code clarity and preventing implicit conversions in some contexts.

Example:

```
#include <functional>
#include <iostream>

double divide(int a, int b) { return double(a) / b; }

int main() {
    int result = std::invoke_r<int>(divide, 7, 2); // Result explicitly converted to
    ↪  int
    std::cout << result << '\n'; // prints 3
}
```

13.1.7 `std::to_underlying`

This utility converts an enumerator to its underlying integral type.

- **Motivation:** Provides a safe and standard way to obtain the underlying integer of an `enum class`, which otherwise requires casting.
- **Use Case:** Common in bitmask operations, serialization, or when enums map directly to protocol values.

Example:

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

enum class Color : unsigned char { Red = 1, Green = 2, Blue = 3 };

int main() {
    auto value = std::to_underlying(Color::Green); // unsigned char -> 2
```

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

13.1.8 `std::unreachable`

A utility function used to mark code paths that should never be executed.

- **Purpose:** Serves as a hint to compilers and static analyzers, enabling optimization and diagnostic improvements.
- **Behavior:** If executed, it results in undefined behavior.
- **Use Case:** Used in `switch` statements over exhaustive enums, or in situations where a branch is provably unreachable.

Example:

```
#include <utility>

enum class Mode { Read, Write };

void process(Mode m) {
    switch (m) {
        case Mode::Read:  /*...*/ break;
        case Mode::Write: /*...*/ break;
    }
    std::unreachable(); // no other values possible
}
```

13.1.9 Summary

The utilities introduced in C++23 greatly expand the expressiveness of the language while addressing practical problems developers face:

- **Error handling:** `std::expected`.
- **Callable handling:** `std::move_only_function`, `std::bind_back`, `std::invoke_r`.
- **Low-level operations:** `std::byteswap`, `std::unreachable`.
- **Forwarding and type conversion:** `std::forward_like`, `std::to_underlying`.

Together, these utilities make C++ safer, clearer, and more powerful, reinforcing its role as both a systems-level and high-level application language.

Chapter 14

Coroutines and `std::generator`

Coroutines were formally introduced in C++20, providing a mechanism to suspend and resume functions at well-defined points. This capability laid the foundation for asynchronous programming, lazy evaluation, and custom control-flow abstractions. While C++20 provided the low-level building blocks of coroutines (such as `co_await`, `co_yield`, and `co_return`), it did not include a ready-to-use, standardized coroutine type for common patterns like generators.
C++23 fills this gap by introducing **`std::generator`**, a standard-library facility that simplifies the creation of coroutines producing sequences of values. This addition bridges the gap between the raw coroutine primitives of C++20 and practical applications, allowing developers to express lazily generated sequences in a concise and type-safe manner.

14.1 Background: Coroutines in C++

A coroutine is a function that can **suspend execution** at certain points and **resume** later, maintaining its local state across suspensions. In C++, coroutines are expressed

using three new keywords:

- `co_return` — returns a value from a coroutine.
- `co_yield` — produces a value and suspends execution.
- `co_await` — suspends until the awaited object is ready.

The compiler transforms coroutine functions into state machines behind the scenes. While this is powerful, writing coroutine ”promise types” and ”awaiter objects” by hand can be complex.
In C++20, the language gave programmers the machinery but no ready-made, standardized coroutine return types. Developers often had to rely on third-party libraries or implement their own wrappers for common tasks.
C++23's `std::generator` provides the first standardized coroutine return type designed specifically for **synchronous generators**.

14.2 The Role of `std::generator`

`std::generator<T>` is a coroutine type that produces a sequence of values of type `T` on demand. It integrates seamlessly with the **range-based for loop** and standard library algorithms, making it both intuitive and efficient.

- **Lazy Evaluation:** Values are generated as needed, avoiding pre-computation or storage of entire sequences.
- **Range Support:** `std::generator` models the standard range concepts, making it compatible with the Ranges library introduced in C++20.
- **Memory Efficiency:** Only one element is produced at a time, minimizing memory usage when dealing with potentially large or infinite sequences.

14.3 Using `std::generator`

The core usage pattern involves writing a coroutine that returns `std::generator<T>` and uses `co_yield` to produce values one at a time.

14.3.1 Example 1: Simple Sequence Generator

```
#include <generator>
#include <iostream>

std::generator<int> numbers_up_to(int n) {
    for (int i = 1; i <= n; ++i) {
        co_yield i;
    }
}

int main() {
    for (int value : numbers_up_to(5)) {
        std::cout << value << " ";
    }
    // Output: 1 2 3 4 5
}
```

In this example, the coroutine `numbers_up_to` lazily produces integers from 1 to `n`. The range-based for loop requests values one at a time, suspending and resuming the coroutine as needed.

14.3.2 Example 2: Infinite Generator

Generators can produce infinite sequences because values are computed lazily.

```
#include <generator>
#include <cstdint>

std::generator<std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield a;
        auto temp = a + b;
        a = b;
        b = temp;
    }
}
```

Consumers can stop iteration at any time, avoiding infinite loops.

```
#include <iostream>

int main() {
    int count = 0;
    for (auto v : fibonacci()) {
        if (count++ == 10) break;
        std::cout << v << " ";
    }
    // Output: 0 1 1 2 3 5 8 13 21 34
}
```

14.4 Integration with Ranges

One of the most powerful aspects of `std::generator` is that it models the **input range concept**, enabling direct integration with `<ranges>` algorithms.

14.4.1 Example: Filtering and Transforming with Ranges

```
#include <generator>
#include <ranges>
#include <iostream>

std::generator<int> even_numbers(int limit) {
    for (int i = 0; i <= limit; ++i) {
        if (i % 2 == 0) co_yield i;
    }
}

int main() {
    auto evens = even_numbers(20)
               | std::views::transform([](int x){ return x * x; });

    for (int v : evens) {
        std::cout << v << " ";
    }
    // Output: 0 4 16 36 64 100 144 196 256 324 400
}
```

By combining `std::generator` with the Ranges library, developers can build pipelines of transformations and filters while generating values lazily.

14.5 Technical Details

- **Header:** `<generator>`
- **Template:** `template<class Ref, class V = void, class Allocator = void> class generator;`

- **Reference Type:** The first template parameter `Ref` specifies the type of values yielded.
- **Value Type:** The second parameter `V` allows specifying a value type different from the reference type.
- **Allocator:** The third parameter provides customization of memory allocation for coroutine frames.

14.5.1 Constraints and Guarantees:

- `std::generator` is **move-only**, not copyable.
- It satisfies the **range** and **input_range** concepts.
- Coroutine frames are automatically managed and destroyed when iteration ends.
- Exceptions propagate naturally through the generator.

14.6 Advantages of `std::generator`

1. **Standardized API:** Provides a reliable, portable alternative to third-party generator implementations.
2. **Efficient Iteration:** Produces values on demand, reducing memory consumption.
3. **Range Compatibility:** Works directly with the Ranges library, enabling powerful functional-style programming.
4. **Expressiveness:** Allows developers to express complex iteration patterns more naturally than writing custom iterators.

14.7 Practical Use Cases

- **Lazy Sequences:** Infinite series, primes, Fibonacci numbers.
- **Streaming Data:** Processing log files, network packets, or event streams one at a time.
- **Tree Traversals:** Recursive traversal of data structures like binary trees without explicit stacks.
- **Pipeline Construction:** Feeding data into ranges pipelines with filters and transformations.

14.8 Summary

C++20 introduced the coroutine mechanism but left practical coroutine abstractions to libraries. C++23 builds upon that foundation with `std::generator`, offering a standardized and elegant way to create lazily evaluated sequences.
With its seamless integration into the Ranges framework, `std::generator` allows developers to combine coroutine power with modern functional-style programming, making C++ more expressive and memory-efficient. It closes a significant usability gap in coroutine programming, providing a facility that developers can immediately apply to real-world tasks without needing to implement low-level coroutine machinery themselves.

Chapter 15

Diagnostics

15.1 The new `<stacktrace>` library

C++23 introduces the `<stacktrace>` library, a long-requested feature that brings standardized facilities for capturing and inspecting call stack information at runtime. Historically, developers relied on platform-specific APIs (such as `backtrace()` on POSIX systems or `StackWalk()` on Windows) or third-party libraries to obtain stack traces. The `<stacktrace>` library provides a **portable, type-safe, and convenient** solution within the standard.

15.1.1 Purpose and Motivation

The `<stacktrace>` library is designed to simplify **runtime diagnostics** by providing a structured way to capture, store, and display the program's call stack. This is particularly useful for:

- **Error reporting:** Attaching stack traces to exceptions or log messages.

- **Debugging support:** Capturing program state at the point of failure.
- **Crash analysis:** Providing developers with actionable information when errors occur in production.

15.1.2 Core Components

The library introduces two main classes:

1. `std::stacktrace_entry`
    - Represents a single frame in a stack trace.
    - Provides access to function names, source file names, and line numbers (when available).
    - Information may vary depending on compiler, platform, and symbol availability.
2. `std::stacktrace`
    - Represents a collection (sequence) of `stacktrace_entry` objects.
    - Can be captured at any point using static member functions like `std::stacktrace::current()`.
    - Provides iteration, comparison, and formatted output.

15.1.3 Capturing a Stack Trace

A stack trace can be obtained at any point in program execution:

```
#include <stacktrace>
#include <iostream>

void foo() {
    auto trace = std::stacktrace::current();
    std::cout << trace << '\n';
}

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

- The call to `std::stacktrace::current()` captures the current call stack.
- Printing the trace outputs a formatted representation of function calls.

15.1.4 Inspecting Individual Entries

`std::stacktrace_entry` allows examination of detailed frame-level information:

```
#include <stacktrace>
#include <iostream>

void bar() {
    auto trace = std::stacktrace::current();
    for (auto const& entry : trace) {
        std::cout << "Function: " << entry.description() << '\n';
        std::cout << "Source: "   << entry.source_file() << ":" <<
        ↪  entry.source_line() << '\n';
    }
}
```

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

Key member functions of `std::stacktrace_entry`:

- `description()` – human-readable function or symbol name.
- `source_file()` – name of the source file, if available.
- `source_line()` – line number within the source file, if available.
- `address()` – the raw memory address corresponding to the frame.

15.1.5 Exception Integration

One of the most powerful use cases is attaching a stack trace to exceptions for debugging:

```
#include <stacktrace>
#include <stdexcept>
#include <iostream>

struct traced_exception : std::runtime_error {
    std::stacktrace trace;
    traced_exception(const std::string& msg)
        : std::runtime_error(msg), trace(std::stacktrace::current()) {}
};

int main() {
    try {
        throw traced_exception("Something went wrong");
```

```
    } catch (const traced_exception& e) {
        std::cerr << e.what() << '\n';
        std::cerr << e.trace << '\n';
    }
}
```

This pattern makes it easy to diagnose problems in production without attaching a debugger.

15.1.6 Performance Considerations

- Capturing stack traces incurs runtime cost, especially when symbol resolution is required.
- For performance-sensitive applications, it is advisable to capture stack traces **only when necessary**, such as during error handling or logging.
- Implementations are allowed to optimize the level of detail provided depending on system capabilities.

15.1.7 Portability and Limitations

- Availability of function names, file names, and line numbers depends on compiler, build settings, and whether debug symbols are present.
- Not all platforms can provide full symbolic information at runtime. In such cases, only memory addresses may be available.
- The library is intended for **diagnostics, not security**. It should not be relied upon for protection or sandboxing.

15.1.8 Example: Combining with Logging

The `<stacktrace>` library integrates well with custom logging frameworks:

```
#include <stacktrace>
#include <iostream>

void log_error(const std::string& msg) {
    std::cerr << "Error: " << msg << '\n';
    std::cerr << "Stack trace:\n" << std::stacktrace::current() << '\n';
}

void process() {
    log_error("Failed to process request");
}

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

This approach allows developers to collect actionable diagnostic data without attaching a debugger.

15.1.9 Summary

The `<stacktrace>` library in C++23 represents a major improvement in **diagnostics and debugging support** within the standard library. Its key contributions include:

- **Portable stack traces:** No need for platform-specific APIs.
- **Structured inspection:** Access to functions, files, and lines (when available).
- **Seamless integration:** Works with exceptions and logging frameworks.

- **Ease of use:** Requires minimal code to capture and print traces.

By making stack traces part of the standard library, C++23 strengthens the diagnostic capabilities of the language, helping developers understand and resolve problems more efficiently.

Chapter 16

Ranges and Algorithms

16.1 New Range Adaptors (`zip`, `adjacent`, `chunk_by`, etc.)

The C++20 standard introduced the **Ranges library**, which unified algorithms, views, and iterators into a composable framework for working with collections of data. Views such as `std::views::filter`, `std::views::transform`, and `std::views::take` made it possible to build powerful pipelines that generate and transform ranges lazily.
C++23 expands this framework by introducing several **new range adaptors**. These additions fill gaps in common iteration patterns, making it easier to express algorithms directly in range pipelines without custom code.

16.1.1 Overview of New Adaptors in C++23

C++23 introduces the following new range adaptors:

- **`std::views::zip`** and **`std::views::zip_transform`**

- **`std::views::adjacent`** and **`std::views::adjacent_transform`**
- **`std::views::chunk`** and **`std::views::chunk_by`**
- **`std::views::slide`**
- **`std::views::stride`**
- **`std::views::join_with`**

These adaptors extend the expressive power of range pipelines by allowing element grouping, synchronization of multiple ranges, and more control over iteration steps.

16.1.2 `std::views::zip` and `std::views::zip_transform`

Purpose `zip` combines multiple ranges into a single range of tuples, iterating in lockstep. This is useful when working with parallel sequences of related data.

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

int main() {
    std::vector<int> a = {1, 2, 3};
    std::vector<char> b = {'a', 'b', 'c'};

    for (auto [x, y] : std::views::zip(a, b)) {
        std::cout << x << " -> " << y << '\n';
    }
}
```

Example: Zipping Two Ranges
Output:

```
1 -> a
2 -> b
3 -> c
```

`zip_transform` `zip_transform` applies a function to the zipped elements directly, avoiding the need to unpack tuples manually.

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

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

    auto sums = std::views::zip_transform(
        [](int a, int b) { return a + b; }, x, y);

    for (int v : sums) std::cout << v << " ";
    // Output: 5 7 9
}
```

16.1.3 `std::views::adjacent` and `std::views::adjacent_transform`

Purpose `adjacent<N>` creates a sliding window of size `N` over a range, yielding tuples of consecutive elements.

- `adjacent<2>` is especially useful for pairwise iteration.
- `adjacent_transform` applies a function to each tuple directly.

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

int main() {
    std::vector<int> data = {10, 20, 30, 50};

    for (auto [a, b] : std::views::adjacent<2>(data)) {
        std::cout << (b - a) << " ";
    }
    // Output: 10 10 20
}
```

Example: Pairwise Differences

16.1.4 `std::views::chunk` and `std::views::chunk_by`

`chunk` Splits a range into **subranges of fixed size**.

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

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

    for (auto chunk : std::views::chunk(numbers, 3)) {
```

```
        for (int n : chunk) std::cout << n << " ";
        std::cout << '\n';
    }
}
```

Output:

```
1 2 3
4 5 6
7
```

`chunk_by` Groups elements based on a **binary predicate**. A new chunk starts whenever the predicate returns `false` between consecutive elements.

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

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

    auto chunks = std::views::chunk_by(data,
        [](int a, int b){ return a == b; });

    for (auto group : chunks) {
        for (int v : group) std::cout << v;
        std::cout << " | ";
    }
}
// Output: 11 | 222 | 33 | 1 |
```

16.1.5 `std::views::slide`

Creates overlapping windows of a specified size, moving forward by one element at a time.

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

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

    for (auto window : std::views::slide(nums, 3)) {
        for (int n : window) std::cout << n;
        std::cout << " ";
    }
}
// Output: 123 234
```

16.1.6 `std::views::stride`

Steps through a range by a specified stride value.

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

int main() {
    std::vector<int> nums = {10,20,30,40,50,60};

    for (int n : std::views::stride(nums, 2)) {
        std::cout << n << " ";
```

```
    }
}
// Output: 10 30 50
```

16.1.7 `std::views::join_with`

Joins a range of ranges, inserting a delimiter between each inner range.

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

int main() {
    std::vector<std::vector<std::string>> words = {
        {"Hello"}, {"C++23"}, {"Ranges"}
    };

    auto joined = std::views::join_with(words, std::string(" "));

    for (const auto& w : joined) std::cout << w;
    // Output: Hello C++23 Ranges
}
```

16.1.8 Summary

The new range adaptors in C++23 significantly expand the expressiveness of the Ranges library:

- **`zip` and `zip_transform`:** Synchronize multiple ranges.

- **`adjacent` and `adjacent_transform`:** Provide sliding windows of consecutive elements.
- **`chunk` and `chunk_by`:** Group elements by size or predicate.
- **`slide`:** Overlapping windows of fixed size.
- **`stride`:** Sample ranges with steps.
- **`join_with`:** Flatten and join ranges with delimiters.

Together, these adaptors bring C++ ranges closer to high-level, declarative data manipulation found in functional languages, while retaining the efficiency and laziness expected from C++.

16.2 New Constrained Algorithms (`find_last`, `starts_with`, etc.)

The C++ standard library has long provided a rich set of algorithms for operating on ranges of data, starting with the `<algorithm>` header. With C++20, these algorithms were redefined in terms of **ranges** and constrained using concepts, ensuring more precise overload resolution and improved safety.
C++23 continues this evolution by adding **new constrained algorithms** that address common usage patterns not previously covered, further aligning the standard with real-world programming needs. These algorithms enhance both expressiveness and correctness, and they integrate seamlessly with the Ranges framework.

16.2.1 Motivation for New Algorithms

Before C++23, developers often had to manually compose existing algorithms, iterators, or loops to achieve tasks such as:

- Finding the last occurrence of an element.
- Checking whether a sequence starts or ends with a particular subsequence.
- Locating the first subsequence that differs from another.

C++23 introduces new constrained algorithms to handle these tasks directly, improving readability, correctness, and maintainability.

16.2.2 Overview of Newly Added Algorithms

The key new constrained algorithms in C++23 include:

- **`std::ranges::find_last`**
- **`std::ranges::starts_with`**
- **`std::ranges::ends_with`**
- **`std::ranges::contains`**
- **`std::ranges::contains_subrange`**

All these algorithms are **constrained with concepts** (e.g., `forward_range`, `input_range`, `sentinel_for`) to ensure they work correctly with ranges and iterators in modern C++.

16.2.3 `std::ranges::find_last`

Purpose Finds the **last occurrence** of a value or element satisfying a predicate in a range.

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

int main() {
    std::vector<int> data = {1, 3, 4, 7, 8, 5};

    auto it = std::ranges::find_last(data, 8);
    if (it != data.end())
        std::cout << "Found: " << *it << '\n';
}
```

Example: Last Even Number

```
auto it = std::ranges::find_last_if(data,
    [](int x) { return x % 2 == 0; });

if (it != data.end())
    std::cout << "Last even number: " << *it << '\n';
```

Example: With Predicate

This removes the need to manually reverse a range or use reverse iterators.

16.2.4 `std::ranges::starts_with`

Purpose Checks whether a range **begins with a given subsequence**.

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

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

    if (std::ranges::starts_with(data, prefix))
        std::cout << "Data starts with the prefix\n";
}
```

Example:

16.2.5 `std::ranges::ends_with`

Purpose Checks whether a range **ends with a given subsequence**.

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

int main() {
    std::vector<int> data = {10,20,30,40,50};
    std::vector<int> suffix = {40,50};

    if (std::ranges::ends_with(data, suffix))
```

```
        std::cout << "Data ends with the suffix\n";
}
```

This is particularly useful for working with strings, file extensions, or tokenized data.

16.2.6 `std::ranges::contains`

Purpose Determines whether a range contains a particular value.

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

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

    if (std::ranges::contains(data, 3))
        std::cout << "Found 3 in data\n";
}
```

This simplifies code that would previously rely on `std::ranges::find` combined with a comparison to `end()`.

16.2.7 `std::ranges::contains_subrange`

Purpose Checks whether one range contains another as a **contiguous subsequence**.

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

```
int main() {
    std::vector<int> data = {1, 2, 3, 4, 5, 6};
    std::vector<int> subseq = {3, 4, 5};

    if (std::ranges::contains_subrange(data, subseq))
        std::cout << "Data contains subsequence 3,4,5\n";
}
```

16.2.8 Advantages of Constrained Algorithms

1. **Readability:** Expresses intent directly (e.g., `starts_with` vs. manually checking).
2. **Correctness:** Reduces risk of off-by-one or iterator misuse errors.
3. **Performance:** Optimized for common cases in standard library implementations.
4. **Range Integration:** Works seamlessly with views and pipelines.

16.2.9 Example: Combined Usage

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

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

    if (std::ranges::starts_with(nums, std::vector{1,2})) {
        std::cout << "Sequence starts with 1,2\n";
    }
```

```
    if (std::ranges::ends_with(nums, std::vector{7,8})) {
        std::cout << "Sequence ends with 7,8\n";
    }

    if (std::ranges::contains(nums, 5)) {
        std::cout << "Sequence contains 5\n";
    }

    if (std::ranges::contains_subrange(nums, std::vector{3,4,5})) {
        std::cout << "Sequence contains subsequence 3,4,5\n";
    }

    auto it = std::ranges::find_last_if(nums, [](int x){ return x % 2 == 0; });
    if (it != nums.end())
        std::cout << "Last even: " << *it << '\n';
}
```

Output:

```
Sequence starts with 1,2
Sequence ends with 7,8
Sequence contains 5
Sequence contains subsequence 3,4,5
Last even: 8
```

16.2.10 Summary

The new constrained algorithms introduced in C++23—`find_last`, `starts_with`, `ends_with`, `contains`, and `contains_subrange`—provide powerful tools for common sequence operations. They improve clarity, reduce boilerplate, and integrate naturally

with the Ranges framework.

- `find_last`: Locates the last occurrence of a value or predicate.
- `starts_with` / `ends_with`: Checks prefixes and suffixes.
- `contains`: Simplifies membership testing.
- `contains_subrange`: Detects contiguous subsequences.

These enhancements make the **algorithms library more expressive and convenient**, encouraging developers to rely on standardized, constrained solutions rather than ad hoc loops.

16.3 `ranges::to` Conversion Utilities

One of the most useful additions to the Ranges framework in C++23 is the introduction of **`std::ranges::to`**, a general-purpose utility for **converting ranges into containers**. While C++20 provided composable pipelines of range adaptors and algorithms, there was no standard, direct way to ”materialize” a range back into a container of a desired type without verbose syntax.
C++23 addresses this gap with `ranges::to`, which provides a clean, standardized mechanism to **collect ranges into containers**, greatly simplifying code that bridges the functional pipeline style with conventional container storage.

16.3.1 1. Motivation

In C++20, consider transforming a range and then storing the results in a `std::vector`. Developers had to write:

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

int main() {
    auto range = std::views::iota(1, 6) | std::views::transform([](int x){ return x *
    ↪  x; });

    std::vector<int> result;
    std::ranges::copy(range, std::back_inserter(result));
}
```

This works but is verbose and requires explicitly preparing a container and inserting elements.

With C++23's `ranges::to`, the same operation becomes concise and expressive:

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

int main() {
    auto result = std::views::iota(1, 6)
                | std::views::transform([](int x){ return x * x; })
                | std::ranges::to<std::vector>();

    for (int v : result) std::cout << v << " ";
    // Output: 1 4 9 16 25
}
```

16.3.2 2. Basic Usage

16.3.2.1 Converting to a Container

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

int main() {
    auto data = std::views::iota(1, 5);

    auto v = std::ranges::to<std::vector<int>>(data);
    auto l = std::ranges::to<std::list<int>>(data);
}
```

Here, `v` is a `std::vector<int>` containing `{1,2,3,4}` and `l` is a `std::list<int>` with the same values.

16.3.2.2 Implicit Deduction of Value Type

`ranges::to` can deduce the value type from the input range when possible:

```
auto v = std::views::iota(1, 4) | std::ranges::to<std::vector>();
// deduces std::vector<int>
```

16.3.3 3. Advanced Conversions

16.3.3.1 Converting into Associative Containers

```
#include <ranges>
#include <set>
#include <iostream>
```

```
int main() {
    auto data = {5, 3, 1, 4, 2};

    auto s = data | std::ranges::to<std::set>();
    for (int x : s) std::cout << x << " ";
    // Output: 1 2 3 4 5
}
```

Since `std::set` maintains ordering and uniqueness, the result is sorted automatically.

16.3.3.2 Converting into `std::map`

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

int main() {
    std::vector<std::pair<int, std::string>> data = {
        {1, "one"}, {2, "two"}, {3, "three"}
    };

    auto m = data | std::ranges::to<std::map<int, std::string>>();
    for (auto& [k,v] : m) std::cout << k << ":" << v << " ";
    // Output: 1:one 2:two 3:three
}
```

16.3.4 4. Pipelining with Views

`ranges::to` is designed to work as the **final stage in a range pipeline**, collecting the transformed data:

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

int main() {
    auto evens = std::views::iota(1, 11)
               | std::views::filter([](int x){ return x % 2 == 0; })
               | std::ranges::to<std::vector>();

    for (int e : evens) std::cout << e << " ";
    // Output: 2 4 6 8 10
}
```

This pattern matches functional languages and modern frameworks where ”lazy pipelines” culminate in a ”collect” operation.

16.3.5 5. Custom Container Construction

`ranges::to` also supports passing constructor arguments to containers:

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

int main() {
    auto squares = std::views::iota(1, 6)
                 | std::views::transform([](int x){ return x * x; })
```

```
                | std::ranges::to<std::vector<int>>(std::allocator<int>{});

    for (int s : squares) std::cout << s << " ";
    // Output: 1 4 9 16 25
}
```

Here, a custom allocator is passed to the vector constructor.

16.3.6 6. Integration with Algorithms

Since `ranges::to` produces a standard container, the result can seamlessly be used with existing algorithms:

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

int main() {
    auto data = std::views::iota(1, 10)
              | std::views::filter([](int x){ return x % 3 == 0; })
              | std::ranges::to<std::vector>();

    std::ranges::reverse(data);

    for (int v : data) std::cout << v << " ";
    // Output: 9 6 3
}
```

16.3.7 7. Benefits of `ranges::to`

1. **Conciseness:** Eliminates boilerplate of constructing containers manually.
2. **Expressiveness:** Integrates naturally as the terminal step in pipelines.
3. **Safety:** Constrained and type-checked, reducing accidental misuse.
4. **Flexibility:** Works with all standard containers and custom ones with suitable constructors.
5. **Efficiency:** Allows implementations to optimize container construction from ranges.

16.3.8 8. Summary

The addition of **`std::ranges::to`** in C++23 fills a major usability gap in the Ranges library. It serves as a standardized and efficient way to:

- Convert views and ranges into containers (`vector`, `list`, `set`, `map`, etc.).
- Collect the results of pipelines without verbose boilerplate.
- Pass constructor arguments when initializing containers.

This utility is now the idiomatic way to bridge **lazy views** with **eager containers**, making range-based programming in C++23 significantly more ergonomic and expressive.

Chapter 17

Containers

17.1 `std::mdspan` (Multidimensional Span)

17.1.1 Introduction

C++23 introduces **`std::mdspan`**, a powerful abstraction for handling **multidimensional arrays** without the overhead of dynamic allocation or the rigidity of fixed-size containers. It generalizes the concept of `std::span` (introduced in C++20) to **N dimensions**, providing a lightweight, non-owning view over contiguous or strided data.

This addition is especially valuable in domains such as **scientific computing, graphics, high-performance computing (HPC), and machine learning**, where efficient handling of multidimensional data (e.g., matrices, tensors) is crucial.

Unlike `std::vector` or `std::array`, `mdspan` does not own its underlying storage—it provides a *view* with customizable layouts and access patterns, enabling both flexibility and efficiency.

17.1.2 Motivation

Before `std::mdspan`, developers had limited standardized options for working with multidimensional data:

- **Raw pointers and manual indexing:** Flexible but error-prone and lacking abstraction.
- **Nested `std::vector` or `std::array`:** Safe but with performance drawbacks due to fragmented storage or rigid structure.
- **Third-party libraries (Eigen, Blitz++, Kokkos, etc.):** Widely used but outside the C++ standard library.

`std::mdspan` fills this gap by providing:

- A standard, lightweight view for multidimensional data.
- Customizable memory layouts (row-major, column-major, or user-defined).
- Efficient element access without forcing a specific container type.

17.1.3 Basic Concepts

`std::mdspan` is defined in the header `<mdspan>`.
It is a class template:

```
template<
    class ElementType,
    class Extents,
    class LayoutPolicy = std::layout_right,
    class AccessorPolicy = std::default_accessor<ElementType>
> class mdspan;
```

- **`ElementType`:** Type of the elements (e.g., `int`, `double`).
- **`Extents`:** Encodes the number of dimensions and their sizes (can be dynamic).
- **`LayoutPolicy`:** Determines how multidimensional indices map to linear memory (`std::layout_right` for row-major, `std::layout_left` for column-major).
- **`AccessorPolicy`:** Defines how elements are accessed (default is direct pointer access).

17.1.4 Example: 2D Matrix

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

int main() {
    std::vector<double> data(12);

    // Create a 2D mdspan view: 3 rows × 4 columns
    std::mdspan<double, std::extents<std::size_t, 3, 4>> matrix(data.data());

    // Fill with values
    for (std::size_t i = 0; i < 3; i++) {
        for (std::size_t j = 0; j < 4; j++) {
            matrix(i, j) = i * 10 + j;
        }
    }

    // Print values
    for (std::size_t i = 0; i < 3; i++) {
        for (std::size_t j = 0; j < 4; j++) {
```

```
            std::cout << matrix(i, j) << " ";
        }
        std::cout << "\n";
    }
}
```

Output:

```
0 1 2 3
10 11 12 13
20 21 22 23
```

Here, the `mdspan` acts as a view over the raw `std::vector<double>` storage, interpreting it as a **3×4 matrix**.

17.1.5 Extents and Dynamic Sizes

`std::extents` describes dimensions. Some or all extents can be **dynamic**.

```
using dyn_matrix_t = std::mdspan<
    int,
    std::dextents<std::size_t, 2> // dynamic 2D extents
>;

int main() {
    std::vector<int> storage(6);
    dyn_matrix_t mat(storage.data(), 2, 3); // 2 rows, 3 columns

    mat(0,0) = 1; mat(0,1) = 2; mat(0,2) = 3;
    mat(1,0) = 4; mat(1,1) = 5; mat(1,2) = 6;

    for (std::size_t i = 0; i < 2; i++) {
```

```
        for (std::size_t j = 0; j < 3; j++) {
            std::cout << mat(i,j) << " ";
        }
        std::cout << "\n";
    }
}
```

Output:

```
1 2 3
4 5 6
```

17.1.6 Memory Layouts

The `LayoutPolicy` controls how indices map to memory:

- **`std::layout_right`:** Row-major (C-style arrays).
- **`std::layout_left`:** Column-major (Fortran-style arrays).
- **`std::layout_stride`:** Custom stride-based layout.

```
std::mdspan<int, std::extents<std::size_t, 2, 3>, std::layout_left> mat(ptr);
```

Example: Column-Major Layout

This interprets the same memory differently, allowing interoperability with external libraries (e.g., Fortran).

17.1.7 Use Cases

- **Scientific Computing:** Efficient representation of matrices, tensors, and grids.
- **Graphics and Image Processing:** Handling images as 2D or 3D pixel arrays.
- **Machine Learning:** Representing tensors without fixed container types.
- **Numerical Libraries:** A standard abstraction to replace third-party multidimensional array wrappers.

17.1.8 Integration with Views and Algorithms

Since `mdspan` provides an **array-like indexing operator**, it integrates smoothly with ranges and algorithms. For instance, iteration can be done with standard loops, or with custom adaptors when combined with views.

```
for (std::size_t i = 0; i < mat.extent(0); ++i)
    for (std::size_t j = 0; j < mat.extent(1); ++j)
        process(mat(i, j));
```

17.1.9 Performance Characteristics

- **Zero-overhead abstraction:** Access via `(i,j,...)` is as efficient as manual pointer arithmetic.
- **Non-owning view:** Does not allocate or deallocate memory.
- **Customizability:** Layout and accessor policies allow optimization for specific domains.

17.1.10 Summary

`std::mdspan` is a major step forward for C++ standard containers and views, providing:

- A **standardized multidimensional array view**.
- Support for **static and dynamic extents**.
- Flexible **memory layouts** (row-major, column-major, strided).
- **Zero-overhead access** to underlying storage.
- Applications in HPC, ML, scientific computing, and graphics.

It closes a long-standing gap in the C++ ecosystem by giving developers a standardized, efficient, and flexible way to work with multidimensional data, without depending on external libraries.

17.2 Flat Containers (`flat_map`, `flat_set`)

17.2.1 Introduction

C++23 introduces two new **associative container types**:

- `std::flat_set`
- `std::flat_map`

These are known as **flat containers**, designed to combine the interface of ordered associative containers (`std::set`, `std::map`) with the storage and iteration performance characteristics of sequence containers like `std::vector`.

Instead of being implemented as balanced binary search trees (like `std::set` and `std::map`), flat containers store their elements in a **contiguous sorted array** (typically a `std::vector`). This design provides:

- Faster iteration due to cache locality.
- Lower memory overhead (no node pointers).
- Trade-offs in insertion and erasure complexity compared to tree-based containers.

17.2.2 Motivation

Traditional ordered associative containers (`std::set`, `std::map`) provide **logarithmic complexity** for insertion, lookup, and erasure, thanks to their tree-based implementation. However, they suffer from:

- Poor cache locality.
- Higher memory overhead (each node requires dynamic allocation and pointers).
- Slower iteration compared to contiguous containers.

Flat containers solve these issues by:

- Storing elements contiguously in sorted order.
- Offering fast iteration (similar to `std::vector`).
- Providing efficient lookup via binary search.

They are particularly effective when:

- The dataset is relatively small to medium-sized.
- Insertions and erasures are infrequent compared to lookups and iterations.
- Cache efficiency is critical, such as in high-performance or embedded systems.

17.2.3 `std::flat_set`

`flat_set` behaves like `std::set` but stores elements in a sorted vector.

Key Properties

- Elements are **unique**.
- Ordered according to a comparator (default is `std::less`).
- Provides logarithmic lookup via binary search.
- Insertions require maintaining sorted order, which may involve shifting elements.

```
#include <flat_set>
#include <iostream>

int main() {
    std::flat_set<int> fs = {5, 1, 3};

    fs.insert(2);
    fs.insert(4);

    for (int x : fs) {
        std::cout << x << " ";
    }
}
```

Example:

Output:

```
1 2 3 4 5
```

17.2.4 `std::flat_map`

`flat_map` is the flat counterpart of `std::map`.

Key Properties

- Stores **key-value pairs** as sorted contiguous data.
- Keys are unique, and ordering is maintained.
- Provides efficient lookup via binary search.
- Insertions and deletions require shifting elements.

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

int main() {
    std::flat_map<int, std::string> fm = {
        {3, "three"}, {1, "one"}
    };

    fm.insert({2, "two"});
    fm[4] = "four"; // inserts if not present

    for (auto& [k, v] : fm) {
        std::cout << k << ":" << v << " ";
```

```
    }
}
```

Example:

Output:

```
1:one 2:two 3:three 4:four
```

17.2.5 Complexity Characteristics

Operation	`std::set` / `std::map` (tree-based)	`std::flat_set` / `std::flat_map` (flat)
Lookup	O(log n)	O(log n) (binary search)
Iteration	O(n), but cache-inefficient	O(n), cache-friendly (contiguous)
Insertion	O(log n) + O(1) (amortized)	O(log n + n) (binary search + shifting)
Erasure	O(log n)	O(log n + n) (binary search + shifting)
Memory overhead	High (nodes, pointers)	Low (contiguous storage)

Key Trade-off:

- Flat containers excel in **read-mostly workloads**, where lookups and iteration dominate.

- They are less suitable for highly dynamic workloads with frequent insertions and deletions.

17.2.6 When to Use Flat Containers

Flat containers are particularly advantageous in:

- **Embedded systems:** where memory efficiency and predictable iteration performance matter.
- **High-performance applications:** where cache locality drastically improves speed.
- **Lookup-heavy scenarios:** where the cost of occasional insertions is offset by faster queries.
- **Sorted small-to-medium datasets:** where shifting elements is not prohibitively expensive.

They are less suitable when:

- The dataset is extremely large with frequent modifications.
- Constant-time insert/erase operations are required.

17.2.7 Example: Comparing `map` vs `flat_map`

```
#include <map>
#include <flat_map>
#include <iostream>
#include <chrono>
```

```cpp
#include <string>

int main() {
    constexpr int N = 100000;

    std::map<int, std::string> m;
    std::flat_map<int, std::string> fm;

    for (int i = 0; i < N; i++) {
        m[i] = "value";
        fm[i] = "value";
    }

    auto start = std::chrono::high_resolution_clock::now();
    volatile auto it = m.find(N/2);
    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "map lookup: "
              << std::chrono::duration<double, std::micro>(end - start).count() << "
              ↪  µs\n";

    start = std::chrono::high_resolution_clock::now();
    volatile auto it2 = fm.find(N/2);
    end = std::chrono::high_resolution_clock::now();
    std::cout << "flat_map lookup: "
              << std::chrono::duration<double, std::micro>(end - start).count() << "
              ↪  µs\n";
}
```

This illustrates that while both offer logarithmic lookups, `flat_map` often performs better due to better cache locality.

17.2.8 Summary

`std::flat_set` and `std::flat_map` expand the range of associative containers available in the C++ standard library by offering:

- **Contiguous storage** for improved iteration performance.
- **Lower memory overhead** compared to tree-based containers.
- **Binary search lookup** with logarithmic complexity.
- **Trade-offs** in insertion and deletion cost due to element shifting.

They are ideal for **read-mostly datasets** and performance-critical applications where cache locality is a bottleneck. By including these containers, C++23 acknowledges the need for specialized container types tailored to modern workloads, bridging the gap between traditional associative containers and sequence containers.

17.3 Improvements to `span` and `string_view`

C++23 continues the evolution of **non-owning view types**, particularly **`std::span`** and **`std::string_view`**, which were first introduced in C++20 and C++17 respectively. These types provide lightweight, **non-owning views over contiguous data**, enabling safer and more expressive code without the overhead of copying. C++23 introduces **significant improvements** in these types, focusing on **usability, interoperability, and compile-time guarantees**.

17.3.1 `std::span` Improvements

`std::span` represents a view over a contiguous sequence of objects, providing array-like access without owning the data. The key improvements in C++23 include:

- **a. `first`, `last`, and `subspan` constexpr evaluations**
  - C++23 allows `std::span::first`, `std::span::last`, and `std::span::subspan` to be **`constexpr` in more contexts**, making it possible to compute subspans at compile time when working with **static data**.

```
#include <span>
#include <array>

constexpr std::array<int,5> arr = {1,2,3,4,5};
constexpr std::span<const int> s(arr);

constexpr auto first3 = s.first<3>();  // compile-time evaluation
static_assert(first3.size() == 3);
```

This allows spans to participate in **compile-time programming**, enabling more efficient embedded and template-heavy code.

- **b. `empty` and `size_bytes` enhancements**
  - `size_bytes()` can now be **`constexpr`**, allowing calculations of total byte size at compile-time.
  - `empty()` is now more **robust in constexpr contexts**.

```
constexpr std::span<int> s2(arr);
static_assert(s2.size_bytes() == sizeof(int) * 5);
```

- **c. Deduction guides and heterogeneous construction**

- C++23 enhances **deduction guides** for constructing spans from arrays, `std::array`, and pointer-size pairs.
- Interoperability with **multidimensional `mdspan`** is improved, allowing easy slicing and view creation.

- **d. Integration with `ranges::to`**
  - `std::span` can now be a target of `std::ranges::to` conversions, making it easier to transform views or pipelines into spans without manual copying.

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

std::vector<int> v = {1,2,3,4};
auto s = v | std::ranges::to<std::span>(); // span view over vector
```

17.3.2 `std::string_view` Improvements

`std::string_view` represents a **non-owning view over character sequences**. C++23 brings the following improvements:

- **a. `starts_with`, `ends_with`**
  - C++23 adds **member functions** and **free functions** for checking prefixes and suffixes.
  - Both `std::string_view` and `std::ranges::starts_with` / `ends_with` support these operations, improving clarity.

```cpp
#include <string_view>
#include <iostream>

int main() {
    std::string_view sv = "C++23 Standard";

    if (sv.starts_with("C++")) {
        std::cout << "Prefix matches\n";
    }

    if (sv.ends_with("Standard")) {
        std::cout << "Suffix matches\n";
    }
}
```

- **b. `remove_prefix` and `remove_suffix` in constexpr contexts**
  - Previously only usable at runtime, now `remove_prefix()` and `remove_suffix()` are **constexpr**, enabling compile-time manipulation of string views.

```cpp
constexpr std::string_view sv = "HelloWorld";
constexpr auto sv2 = sv.substr(0,5); // "Hello"
```

- **c. Interoperability with ranges**
  - `std::string_view` now works seamlessly with **ranges algorithms** in C++23, e.g., `std::ranges::starts_with`, `std::ranges::contains_subrange`.

– This allows pipeline-style processing without copying strings.

```
#include <ranges>
#include <string_view>
#include <iostream>

int main() {
    std::string_view text = "C++23 spans and views";
    if (std::ranges::contains_subrange(text, "spans")) {
        std::cout << "Subrange found\n";
    }
}
```

- **d. Deduction guides for string literals**

  – String literals now **deduce `std::string_view` automatically** in more contexts, simplifying initialization in constexpr and template-heavy code.

```
constexpr std::string_view greeting = "Hello, C++23!";
```

17.3.3 Benefits of These Improvements

1. **Compile-time operations:** Many functions (`first`, `subspan`, `remove_prefix`, `remove_suffix`) are now constexpr, improving safety and performance.
2. **Improved interoperability:** Works naturally with `ranges` and pipelines.
3. **Expressiveness:** New member functions (`starts_with`, `ends_with`) reduce boilerplate and improve readability.

4. **Efficiency:** No copying of data; all operations are views or lightweight manipulations.

5. **Consistency:** Harmonizes `span` and `string_view` with the modern C++23 ranges and containers ecosystem.

17.3.4 Example: Combined Use

```
#include <span>
#include <string_view>
#include <ranges>
#include <iostream>

int main() {
    constexpr std::array<int, 5> arr = {10, 20, 30, 40, 50};
    constexpr std::span<const int> s(arr);
    constexpr auto last2 = s.last<2>(); // {40,50}

    for (int v : last2) std::cout << v << " ";
    std::cout << "\n";

    constexpr std::string_view sv = "C++23 Modern Library";
    if (sv.starts_with("C++")) std::cout << "Valid prefix\n";
    if (sv.ends_with("Library")) std::cout << "Valid suffix\n";
}
```

Output:

```
40 50
Valid prefix
Valid suffix
```

This example demonstrates **compile-time slicing with `span`** and **string inspection with `string_view`**, illustrating how C++23 makes these views more versatile and efficient.

17.3.5 Summary

C++23 enhances `std::span` and `std::string_view` to make them **more constexpr-friendly, more interoperable with ranges, and more expressive**. Key highlights:

- `span`: improved `constexpr` support, subspan manipulation, integration with `ranges::to`.
- `string_view`: `starts_with`, `ends_with`, `remove_prefix/suffix` in `constexpr`, seamless ranges integration.
- Both types remain **lightweight, non-owning views** that reduce copying and improve performance.

These improvements make `span` and `string_view` **central tools for modern C++23 programming**, especially in **high-performance, template-heavy, and embedded systems** contexts.

Chapter 18

Compile-Time Library Additions

18.1 `constexpr` Expansions (`std::bitset`, `unique_ptr`, etc.)

C++23 continues the trend initiated in C++20 by **expanding `constexpr` capabilities across the standard library**, enabling more operations to be evaluated at compile-time. This expansion allows developers to **perform complex computations, container manipulations, and resource management in `constexpr` contexts**, significantly improving **compile-time correctness, performance, and safety**.
In this section, we explore the key `constexpr` expansions, focusing on **`std::bitset`**, **`std::unique_ptr`**, and other standard library types.

18.1.1 `std::bitset constexpr` Enhancements

`std::bitset` represents a fixed-size sequence of bits and is commonly used for bit manipulation.

Key C++23 Improvements

1. **All member functions are `constexpr`**:
    - Creation, assignment, flipping, setting, resetting, and counting bits can now occur at compile-time.
2. **Compile-time bitwise operations**:
    - Logical operations (`&`, `|`, `^`, `~`) are now fully usable in `constexpr` contexts.

```cpp
#include <bitset>
#include <iostream>

constexpr std::bitset<8> make_mask() {
    std::bitset<8> mask;
    mask.set(1);
    mask.set(3);
    mask.flip(2);
    return mask;
}

constexpr auto mask = make_mask();

static_assert(mask[1] == true);
static_assert(mask[2] == true);  // flipped
```

Example: Compile-Time Bitset

This enables **compile-time validation** of bit patterns, useful for embedded programming, protocol design, and compile-time configuration.

18.1.2 `std::unique_ptr` in `constexpr` Contexts

`std::unique_ptr` is now partially usable in **`constexpr` functions**, enabling **resource management at compile-time** when the allocator and deletion mechanisms allow it.

Highlights

- Default deleter and raw pointers can now be manipulated in `constexpr`.
- `get()`, `release()`, `reset()` and dereference operators are usable in `constexpr` functions.

```
#include <memory>

struct Node {
    int value;
};

constexpr Node* make_node() {
    auto ptr = std::unique_ptr<Node>(new Node{42});
    return ptr.release();  // compile-time transfer of ownership
}

constexpr Node* n = make_node();
static_assert(n->value == 42);
```

Example: `unique_ptr` at Compile-Time

> Note: Full support depends on whether the **allocation is allowed in a constant expression**, which is implementation-defined.

18.1.3 `std::vector` and `std::string` `constexpr` Enhancements

C++23 extends `constexpr` support to **resizable containers** (partially following C++20 improvements):

- `std::vector` and `std::string` now allow **element insertion, resizing, and indexing at compile-time** when the underlying allocator supports it.
- `push_back`, `emplace_back`, `operator[]`, and `size()` can be used in `constexpr` contexts.

```
#include <vector>
#include <array>

constexpr std::vector<int> make_vector() {
    std::vector<int> v;
    v.push_back(10);
    v.push_back(20);
    return v;
}

constexpr auto v = make_vector();
static_assert(v[1] == 20);
```

Example: Compile-Time Vector

This improvement allows **compile-time construction of containers with dynamic behavior**, bridging the gap between fixed arrays and full runtime containers.

18.1.4 Other Standard Library Types

C++23 continues to expand `constexpr` across many other types:

- **`std::array`**: Already fully `constexpr` in C++20; now benefits from better integration with other `constexpr` algorithms.
- **`std::optional`**: All modifying operations (`emplace`, `reset`) are `constexpr`.
- **`std::variant`**: Assignment, emplace, and visitation functions are usable at compile-time.
- **`std::string_view`**: Substring operations, comparison, and search functions are `constexpr`.
- **`std::bitset`**: Already discussed, fully `constexpr` now.

These expansions allow developers to **perform sophisticated computations, type manipulations, and resource initialization at compile-time**, significantly enhancing **performance, safety, and compile-time diagnostics**.

18.1.5 Benefits of `constexpr` Expansions

1. **Compile-time correctness:** Many errors can now be caught at compile-time instead of runtime.
2. **Performance improvements:** Computations moved to compile-time reduce runtime overhead.
3. **Enhanced template programming:** Enables more complex `constexpr` algorithms and template metaprogramming.
4. **Improved embedded and safety-critical programming:** Compile-time resource management and bit manipulations reduce runtime risks.

18.1.6 Example: Combined Compile-Time Usage

```
#include <bitset>
#include <memory>
#include <array>

constexpr std::bitset<8> create_mask() {
    std::bitset<8> mask;
    mask.set(0);
    mask.set(7);
    return mask;
}

constexpr auto mask = create_mask();
static_assert(mask.count() == 2);

constexpr std::array<int, 3> arr = {1,2,3};
constexpr auto first_element = arr[0];
static_assert(first_element == 1);
```

This demonstrates how **bitsets, arrays, and other types can now participate in fully `constexpr` computations**, enabling powerful compile-time logic.

18.1.7 Summary

C++23 significantly expands **`constexpr` support** across the standard library, including:

- **`std::bitset`**: full compile-time bit manipulation.
- **`std::unique_ptr`**: partial compile-time dynamic allocation.

- **Resizable containers**: `std::vector` and `std::string` partially usable at compile-time.
- **Other types**: `optional`, `variant`, `array`, `string_view` fully or partially `constexpr`.

These enhancements make **compile-time programming more practical, expressive, and powerful**, bridging the gap between static and dynamic computations in modern C++23.

18.2 New Type Traits

C++23 expands the **type traits library** (`<type_traits>`) to provide **more expressive, precise, and convenient compile-time introspection** of types. These new traits enable developers to write **safer, more generic, and highly optimized template code**, leveraging C++23's focus on **compile-time correctness and efficiency**.

18.2.1 Motivation

Type traits have long been a core part of C++ template metaprogramming. They allow developers to:

- Inspect properties of types (`is_integral`, `is_pointer`)
- Transform types (`remove_const`, `add_pointer`)
- Enable or disable templates via `std::enable_if` or concepts

C++23 introduces new type traits to **fill gaps in earlier standards**, especially for:

- **Conditional type transformations**
- **More precise detection of constructibility, convertibility, and triviality**
- **Compile-time introspection of function properties**

These traits improve both **readability** and **compile-time safety**, reducing the need for complex SFINAE patterns.

18.2.2 Key New Type Traits

- **a. `std::is_scoped_enum`**

  Detects whether a type is a **scoped enumeration** (declared with `enum class`) versus an unscoped `enum`.

```
enum class Color { Red, Green, Blue };
enum OldEnum { A, B, C };

static_assert(std::is_scoped_enum_v<Color>);
static_assert(!std::is_scoped_enum_v<OldEnum>);
```

  - Useful for templates that need to differentiate **strongly typed enums** from traditional enums.

- **b. `std::is_bounded_array` and `std::is_unbounded_array`**

  Detects whether an array has a **fixed size** or is **dynamic (unknown size)**.

```
int arr1[5];
int arr2[];
```

```
static_assert(std::is_bounded_array_v<decltype(arr1)>);
static_assert(std::is_unbounded_array_v<decltype(arr2)>);
```

  - Improves generic programming involving **array bounds deduction**.

- **c. `std::remove_cvref`**

  Removes **const**, **volatile**, and **reference qualifiers** in a single step, simplifying template code.

```
using T = const volatile int&;
using U = std::remove_cvref_t<T>;  // int
```

  - Before C++23, this required chaining `std::remove_cv` and `std::remove_reference`.
  - Improves **code clarity** in generic templates.

- **d. `std::type_identity` improvements**

  `std::type_identity` now supports **constexpr and more flexible usage**, allowing it to preserve types in complex template manipulations.

```
template<typename T>
using preserve_t = typename std::type_identity<T>::type;
```

  - Useful in **template metaprogramming**, e.g., delaying type evaluation or forwarding types without modification.

- **e. `std::is_invocable_r` and `std::invoke_r` improvements**

  C++23 clarifies and extends traits for detecting **callable objects** with specific return types:

```
#include <type_traits>
#include <functional>

auto lambda = [](int x){ return x*2; };
static_assert(std::is_invocable_r_v<int, decltype(lambda), int>);
```

  - Allows compile-time checking that a callable can be invoked with certain arguments and return a compatible type.
  - Integrates with **`std::invoke_r`**, which allows forcing a return type for invocations.

- **f. `std::is_constant_evaluated`**

  Provides a trait to detect **whether code is being evaluated in a `constexpr` context**, enabling **compile-time optimizations** or assertions in templates:

```
#include <type_traits>

constexpr int f(int x) {
    if (std::is_constant_evaluated()) return x*2;
    return x*3;
}

static_assert(f(2) == 4); // evaluated at compile-time
```

- Useful in **high-performance libraries** and **embedded systems** where behavior may differ between compile-time and runtime.

18.2.3 Other Useful C++23 Type Traits

- **`std::to_underlying`**: Converts scoped enum to underlying integral type.
- **`std::is_aggregate_v`**: Detects aggregate types, useful in template initialization.
- **`std::is_trivially_copyable` refinements**: More precise detection of trivially copyable types.

18.2.4 Example: Combined Usage

```
#include <type_traits>
#include <iostream>

enum class Color { Red, Green };

template<typename T>
void inspect_type() {
    if constexpr(std::is_scoped_enum_v<T>) {
        std::cout << "Scoped enum detected\n";
    } else if constexpr(std::is_arithmetic_v<T>) {
        std::cout << "Arithmetic type\n";
    }
}

int main() {
    inspect_type<Color>();  // Scoped enum detected
    inspect_type<int>();    // Arithmetic type
}
```

This demonstrates how **new type traits enable simpler, clearer, and safer compile-time logic**.

18.2.5 Benefits of New Type Traits

1. **Simplified template code:** Less need for chaining multiple traits or SFINAE.
2. **Improved compile-time checks:** Detect type properties more accurately.
3. **Better generic programming:** Easier to write templates that adapt based on type characteristics.
4. **Integration with modern features:** Works seamlessly with `constexpr`, concepts, and ranges.

18.2.6 Summary

C++23 expands the type traits library to provide:

- `std::is_scoped_enum` for strongly typed enums
- `std::is_bounded_array` and `std::is_unbounded_array` for array detection
- `std::remove_cvref` for simplified type transformations
- Enhanced `std::is_invocable_r` and `std::invoke_r` for callable detection
- Support for `std::is_constant_evaluated` in templates
- Various refinements in `type_identity`, `is_aggregate`, and trivially copyable detection

These enhancements allow **compile-time introspection and manipulation** to be more expressive, precise, and user-friendly, reinforcing **C++23's focus on safer and more efficient compile-time programming**.

Chapter 19

Memory Management Updates

19.1 `std::out_ptr` and `std::inout_ptr`

C++23 introduces **`std::out_ptr`** and **`std::inout_ptr`**, two **helper utilities** designed to simplify **interfacing smart pointers with legacy APIs** that require raw pointer arguments. These utilities improve **safety, readability, and correctness** when working with legacy code that expects **output or in-out raw pointers**.

19.1.1 Motivation

Many C and C++ APIs, especially older or system-level libraries, expect pointers as **output parameters**, for example:

```
int* p = nullptr;
legacy_create(&p); // fills p with dynamically allocated memory
```

When using **smart pointers** (`std::unique_ptr`, `std::shared_ptr`), directly passing raw pointers can be error-prone:

- Requires manual release of ownership before passing.
- Introduces potential memory leaks or double-deletes.
- Leads to verbose and unsafe code.

`std::out_ptr` and `std::inout_ptr` solve this by providing **temporary adapters** that safely extract the raw pointer for legacy APIs while **maintaining smart pointer ownership**.

19.1.2 `std::out_ptr`

`std::out_ptr` is used when a **smart pointer will receive a new object from a legacy API**.

Key Properties

- Works with `std::unique_ptr` and `std::shared_ptr`.
- Provides a **temporary raw pointer** that can be passed to functions expecting `T**` (output).
- Automatically **releases the previous object** (if any) from the smart pointer.
- Ensures that the smart pointer **assumes ownership** of the newly created object.

```
#include <memory>
#include <memory_resource> // for std::out_ptr

struct Resource {
    int value;
```

```
};

// Legacy function: fills a pointer with a new object
void legacy_create(Resource** r) {
    *r = new Resource{42};
}

int main() {
    std::unique_ptr<Resource> res;

    legacy_create(std::out_ptr(res)); // res now owns the new object

    return res->value; // safe access
}
```

Example

Benefits:

- No manual `delete` required.
- Ownership is safely transferred to `unique_ptr`.
- Prevents memory leaks if `res` previously held an object.

19.1.3 `std::inout_ptr`

`std::inout_ptr` is used when a **smart pointer both provides and receives an object**.

Key Properties

- Temporarily converts a smart pointer into a raw `T**` for **functions that may modify the pointer**.

- Ensures **automatic restoration of ownership** in the smart pointer after the call.
- Useful for APIs that **reallocate or update resources** passed via pointer-to-pointer.

```
#include <memory>
#include <iostream>
#include <memory_resource> // for std::inout_ptr

void legacy_reassign(Resource** r) {
    delete *r;
    *r = new Resource{100};
}

int main() {
    std::unique_ptr<Resource> res = std::make_unique<Resource>(42);

    legacy_reassign(std::inout_ptr(res)); // res now points to new object

    std::cout << res->value; // Output: 100
}
```

Example
Benefits:

- Safe temporary access to legacy APIs.
- Automatically cleans up old object if replaced.
- Eliminates manual pointer management and potential leaks.

19.1.4 Key Differences Between `out_ptr` and `inout_ptr`

Feature	**`std::out_ptr`**	**`std::inout_ptr`**
Purpose	Receives a new object from API	API may read & write existing object
Ownership before call	Released (if any)	Preserved until API call completes
Ownership after call	Smart pointer assumes new object	Smart pointer assumes modified/new object
Typical usage	Output parameters (`T**`)	In-out parameters (`T**`)
Example	`std::out_ptr(ptr)`	`std::inout_ptr(ptr)`

19.1.5 Integration with Legacy APIs

These helpers are particularly useful for:

- Windows COM interfaces (`CoCreateInstance` with `IUnknown**`)
- C-style libraries that return dynamically allocated objects via output pointers
- Resource-managing functions in graphics, network, or system libraries

By using `std::out_ptr` and `std::inout_ptr`, developers can **modernize legacy interactions** without sacrificing safety or introducing manual memory management.

19.1.6 Example: COM-style API

```
#include <memory>
#include <iostream>
#include <memory_resource> // C++23 adapters

struct COMObject { int data; };

// Legacy COM-style factory
void create_com_object(COMObject** obj) {
    *obj = new COMObject{2023};
}

int main() {
    std::unique_ptr<COMObject> com_ptr;

    create_com_object(std::out_ptr(com_ptr)); // automatically owns new object

    std::cout << com_ptr->data; // safe access
}
```

This pattern avoids raw pointer errors, **simplifies code**, and ensures **RAII compliance** with smart pointers.

19.1.7 Summary

C++23's `std::out_ptr` and `std::inout_ptr` provide **modern, safe, and convenient adapters** for:

- Interfacing **smart pointers with legacy APIs**.
- Maintaining **RAII semantics** while providing raw pointer access.

- Reducing **memory leaks and manual pointer management**.
- Improving **readability and maintainability** in codebases that mix modern C++ and legacy libraries.

These utilities reflect C++23's focus on **bridging modern resource management with real-world legacy requirements**, making smart pointers **more versatile in practical software development**.

19.2 `allocate_at_least` and `start_lifetime_as`

C++23 introduces **two important memory management utilities** in `<memory>` and `<memory_resource>` that improve **performance, flexibility, and control over object lifetimes**:

1. `allocate_at_least` – allows allocation of **at least a requested number of objects**, giving the allocator freedom to over-allocate for efficiency.
2. `start_lifetime_as` – allows **starting the lifetime of an object in storage of another type**, enabling safe **type punning and memory reuse**.

These utilities reflect C++23's focus on **modern memory control** and **compile-time safety**.

19.2.1 `allocate_at_least`

Overview `allocate_at_least` is an **allocator function** that requests memory for **at least `n` elements of type `T`**, but allows the allocator to provide more memory for performance reasons, such as:

- Reducing the number of allocations for dynamic containers.
- Allowing allocators to optimize memory usage.

```
template<class Alloc>
struct allocator_traits {
    static auto allocate_at_least(Alloc& a, std::size_t n);
};
```

Declaration

- Returns a **pair**: `{pointer, size_type}`.
- `pointer` points to the allocated storage.
- `size_type` is the **number of elements actually allocated**, which may be greater than `n`.

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

int main() {
    std::allocator<int> alloc;

    auto [ptr, count] =
    ↪  std::allocator_traits<std::allocator<int>>::allocate_at_least(alloc, 5);

    std::cout << "Allocated at least 5 ints, actually got: " << count << "\n";

    // Use the allocated memory
    for (std::size_t i = 0; i < 5; ++i) {
        alloc.construct(ptr + i, int(i * 10));
    }
```

```
    for (std::size_t i = 0; i < 5; ++i) {
        std::cout << ptr[i] << " ";
        alloc.destroy(ptr + i);
    }

    alloc.deallocate(ptr, count);
}
```

Example

Benefits:

- Reduces **allocation overhead** when working with containers or memory pools.
- Supports **efficient memory reuse** in high-performance and real-time applications.
- Improves **allocator flexibility** without sacrificing safety.

19.2.2 `start_lifetime_as`

Overview `start_lifetime_as` allows a **new object to begin its lifetime in storage already allocated for another type**. This is particularly useful for:

- **Type punning** – safely reinterpreting memory as a different type.
- **Manual memory reuse** – avoiding repeated allocations for objects of different types in the same memory block.

```
template<class T, class U>
T* start_lifetime_as(U* storage);
```

Declaration

- Converts `storage` into storage suitable for a `T` object.
- Starts the lifetime of a `T` object at the given memory location.
- Ensures **strict aliasing rules are preserved**.

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

struct A { int x; };
struct B { double y; };

int main() {
    alignas(B) char buffer[sizeof(B)];  // raw storage

    B* b_ptr = new(buffer) B{3.14};     // construct B in buffer
    std::cout << b_ptr->y << "\n";

    A* a_ptr = std::start_lifetime_as<A>(reinterpret_cast<A*>(b_ptr));
    new(a_ptr) A{42};                    // safely reuse storage for A
    std::cout << a_ptr->x << "\n";

    a_ptr->~A();  // manual destruction
    b_ptr->~B();  // only necessary if B lifetime still considered active
}
```

Example
Benefits:

- Enables **safe memory reuse**, reducing allocation overhead.
- Respects **object lifetime and strict aliasing rules**, preventing undefined behavior.
- Useful in **custom containers, memory pools, and performance-critical systems**.

19.2.3 Combined Usage

`allocate_at_least` and `start_lifetime_as` can be used together to implement **high-performance memory pools**:

- Allocate a block of memory with `allocate_at_least`.
- Start the lifetime of multiple different objects in that block using `start_lifetime_as`.
- Reuse memory without deallocation between object lifetimes.

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

int main() {
    std::allocator<int> alloc;

    auto [ptr, count] =
    ↪  std::allocator_traits<std::allocator<int>>::allocate_at_least(alloc, 10);
```

```
    for (std::size_t i = 0; i < 10; ++i) {
        int* p = std::start_lifetime_as<int>(ptr + i);
        new(p) int(i * 2);
        std::cout << *p << " ";
        p->~int(); // destroy manually
    }

    alloc.deallocate(ptr, count);
}
```

This example demonstrates **efficient memory allocation and safe object construction/destruction** using the new C++23 facilities.

19.2.4 Summary

C++23 introduces **advanced memory management tools**:

Feature	Purpose
`allocate_at_least`	Allocates memory for at least `n` objects, returning the actual allocation.
`start_lifetime_as`	Starts the lifetime of a new object in memory of another type safely.

Key Advantages:

- Enables **efficient memory reuse** and **high-performance allocations**.
- Reduces **allocation overhead** for containers, pools, and embedded systems.
- Provides **safe lifetime management** respecting C++ object model rules.

- Integrates seamlessly with **modern smart pointers and allocators**.

These utilities are ideal for developers working on **low-level libraries, performance-critical systems, or custom memory management** in C++23.

Chapter 20

String and Text Processing

20.1 `contains`, `resize_and_overwrite`, `substr` Improvements

C++23 introduces **several enhancements to `std::string` and `std::string_view`** that simplify common string operations, improve performance, and reduce boilerplate code. The key features in this section are:

- **`contains`**: a convenient method to check substring presence.
- **`resize_and_overwrite`**: allows in-place modification of string contents efficiently.
- **Improved `substr`**: safer and more expressive substring extraction.

These features enhance **readability, performance, and safety** in string manipulation.

20.1.1 `std::string::contains`

Overview Prior to C++23, checking if a substring exists required using `find` or `std::search`:

```
std::string s = "Hello, C++23";
bool found = s.find("C++") != std::string::npos;
```

C++23 adds a **direct `contains` method**:

```
bool contains(const std::string& s, const std::string& sub);
```

- Returns `true` if `sub` exists in `s`, `false` otherwise.
- Overloaded for `std::string_view` and character literals.
- Improves **readability** and **expressiveness**.

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

int main() {
    std::string text = "Welcome to C++23";

    if (text.contains("C++23")) {
        std::cout << "Found substring!\n";
    }
}
```

Example

Advantages:

- Eliminates the need for `find` comparison with `npos`.
- Makes code **self-explanatory**.
- Works seamlessly with **string views** for zero-copy checks.

20.1.2 `std::string::resize_and_overwrite`

Overview `resize_and_overwrite` allows **resizing a string and modifying its content in place** without creating temporary objects.

```
std::string s;
s.resize_and_overwrite(new_size, [](char* buffer, size_t size) -> size_t {
    // fill buffer up to size
    for (size_t i = 0; i < size; ++i) buffer[i] = 'A' + i;
    return size; // actual length written
});
```

- Accepts a **lambda or callable** that writes into the string's internal buffer.
- The callable returns the **number of characters actually written**, allowing the string to **resize correctly**.
- Eliminates **extra allocations and copies** when building strings dynamically.

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

int main() {
    std::string s;
```

```
    s.resize_and_overwrite(5, [](char* buffer, size_t size) -> size_t {
        for (size_t i = 0; i < size; ++i)
            buffer[i] = 'a' + i;
        return size; // final string length
    });

    std::cout << s; // Output: abcde
}
```

Example

Benefits:

- Efficient for **performance-critical code**.
- Useful in **I/O operations** or **string formatting** without intermediate allocations.
- Integrates safely with **RAII**, avoiding manual buffer management.

20.1.3 `std::string::substr` Improvements

Overview The classic `substr(pos, count)` had **limitations**:

- Could throw exceptions if `pos > size()`.
- Required careful bounds checking to avoid undefined behavior.

C++23 **refines `substr`** to make it:

- Safer: returns an empty string if `pos > size()`.
- More expressive: works seamlessly with **string views**, supporting `constexpr` usage.

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

int main() {
    std::string text = "C++23 Standard Library";

    auto sub1 = text.substr(0, 3);    // "C++"
    auto sub2 = text.substr(100, 5);  // safely returns empty string

    std::cout << sub1 << "\n"; // C++
    std::cout << sub2 << "\n"; // (empty)
}
```

Example

Advantages:

- Reduces need for manual `min` or bounds checks.
- Avoids **exceptions for out-of-range substrings**.
- Works naturally with **compile-time evaluation** when combined with `constexpr` strings.

20.1.4 Combined Example

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

int main() {
    std::string s;
```

```
    // Build string efficiently
    s.resize_and_overwrite(6, [](char* buf, size_t) -> size_t {
        std::copy_n("Modern", 6, buf);
        return 6;
    });

    // Check substring
    if (s.contains("Mod")) {
        std::cout << "Substring found!\n";
    }

    // Extract safe substring
    std::string sub = s.substr(0, 10); // automatically safe
    std::cout << sub << "\n";          // Output: Modern
}
```

This example demonstrates **efficient construction, safe substring access, and easy substring checks**.

20.1.5 Benefits Summary

Feature	Improvement in C++23
`contains`	Direct substring check without `find` and `npos` comparison.
`resize_and_overwr`	Efficient in-place modification and resizing of strings.
`substr`	Safe out-of-bounds handling, improved `constexpr` support.

Overall Advantages:

- **Simpler and more readable code**

- **Improved performance** for dynamic string building
- **Safer substring operations**, reducing exceptions and errors
- **Better integration with `string_view` and `constexpr` programming**

These improvements reflect C++23's focus on **modern, efficient, and safe string manipulation**, making `std::string` and `std::string_view` more **expressive, performant, and robust** for everyday programming tasks.

20.2 Formatting Ranges and Tuples

C++23 introduces **enhanced formatting capabilities** that extend the **`std::format` library** (`<format>`) to support **ranges and tuples directly**, improving expressiveness, convenience, and consistency when outputting collections of data.
These additions make **formatted printing of arrays, vectors, and tuples** more natural, eliminating the need for manual loops or helper functions.

20.2.1 Motivation

Prior to C++23, formatting a container or tuple required manual iteration or custom code:

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

std::vector<int> v = {1, 2, 3};
for (auto i : v) std::cout << i << " ";
```

For tuples, formatted printing was even more cumbersome:

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

auto t = std::make_tuple(1, 2, "text");
// required custom function or structured binding for printing
```

C++23 enhances **`std::format`** to directly support these types, reducing boilerplate and improving **readability**.

20.2.2 Formatting Ranges

A **range** is any container or view supporting **`begin` and `end`** iterators, such as `std::vector`, `std::array`, or `std::ranges::view`.

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

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

    std::string formatted = std::format("Vector contents: {}\n", v);
    std::cout << formatted; // Output: Vector contents: [10, 20, 30]
}
```

Usage

Key Points:

- Ranges are formatted as **comma-separated lists** inside square brackets `[]` by default.

- Works with any **range-compatible container**: `std::vector`, `std::array`, `std::deque`, and `std::ranges::view`.
- Supports **format specifiers** for elements:

```
std::vector<double> v = {1.2345, 6.789};
std::cout << std::format("{:.2f}", v) << "\n"; // [1.23, 6.79]
```

Benefits:

- Eliminates manual loops for formatting output.
- Consistent style across different container types.
- Supports **compile-time formatting** with `constexpr` containers.

20.2.3 Formatting Tuples

Tuples represent **heterogeneous collections**. C++23 allows **direct formatting of tuples**, producing readable output without unpacking.

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

int main() {
    auto t = std::make_tuple(42, 3.14, "C++23");

    std::string s = std::format("Tuple: {}\n", t);
    std::cout << s; // Output: Tuple: (42, 3.14, "C++23")
}
```

Example

Key Points:

- Tuple elements are displayed in **parentheses**, separated by commas.
- Supports **nested tuples** and **tuples containing ranges**:

```
auto nested = std::make_tuple(std::vector<int>{1,2}, std::tuple<int,int>{3,4});
std::cout << std::format("Nested: {}\n", nested);
// Output: Nested: ([1, 2], (3, 4))
```

- Each element can use **format specifiers**:

```
auto t = std::make_tuple(1.234, 5.678);
std::cout << std::format("{:.1f}", t) << "\n"; // (1.2, 5.7)
```

20.2.4 Customization and Extensibility

C++23 allows **custom formatters** for user-defined types stored in ranges or tuples:

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

struct Point { int x, y; };

template<>
struct std::formatter<Point> : std::formatter<std::string> {
    auto format(Point p, auto& ctx) {
```

```
        return std::formatter<std::string>::format(
            "(" + std::to_string(p.x) + "," + std::to_string(p.y) + ")", ctx);
    }
};

int main() {
    std::vector<Point> points = {{1,2},{3,4}};
    std::cout << std::format("Points: {}\n", points); // Points: [(1,2), (3,4)]
}
```

- Makes `std::format` **fully extensible** for custom types in collections.
- Works seamlessly with **nested structures**.

20.2.5 Combined Example: Ranges and Tuples

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

int main() {
    std::vector<int> v = {1,2,3};
    auto t = std::make_tuple("Hello", 42, v);

    std::cout << std::format("Tuple with range: {}\n", t);
    // Output: Tuple with range: ("Hello", 42, [1, 2, 3])
}
```

- Demonstrates **nested formatting**.
- Shows **uniform output** for heterogeneous collections and ranges.

20.2.6 Benefits of C++23 Range and Tuple Formatting

1. **Readability:** Eliminates manual printing loops.
2. **Consistency:** Uniform formatting for all standard containers and tuples.
3. **Performance:** No unnecessary copies; formatted directly from ranges.
4. **Safety:** Works with `string_view`, `constexpr` ranges, and tuples.
5. **Extensibility:** Custom types can define formatters, even inside ranges or tuples.

20.2.7 Summary

C++23 enhances **string and text processing** by allowing:

- **Direct formatting of ranges** (`std::vector`, `std::array`, `std::ranges::view`)
- **Direct formatting of tuples**, including heterogeneous and nested tuples
- **Support for format specifiers** per element
- **Custom formatters** for user-defined types in ranges and tuples

These enhancements make `std::format` **more expressive, readable, and practical** for everyday use in modern C++ programming, significantly reducing boilerplate code and improving maintainability.

20.3 Safer String Construction Rules

C++23 introduces **enhancements to `std::string` and `std::string_view` construction** to improve **safety, correctness, and expressiveness**. These changes

are designed to **prevent common bugs**, such as out-of-bounds access, dangling pointers, or undefined behavior during string initialization, particularly when working with **raw character arrays, pointers, or iterators**.

20.3.1 Motivation

Prior to C++23, constructing strings from **raw pointers or iterators** could result in subtle bugs:

```
char buffer[5] = {'a','b','c','d','e'};
std::string s(buffer, 10); // Undefined behavior: count exceeds buffer size
```

Similarly, using **`std::string_view`** with raw pointers could easily lead to dangling references:

```
std::string_view sv;
{
    std::string temp = "hello";
    sv = temp; // sv now dangling after temp is destroyed
}
```

C++23 addresses these issues by **enforcing safer rules during construction**, while maintaining performance and flexibility.

20.3.2 Key Rules for Safer Construction

C++23 introduces the following **guidelines and changes**:

1. **Bounds Checking**
    - When constructing a `std::string` or `std::string_view` from a **pointer and length**, the length must not exceed the actual allocated range.

- Implementations **may perform checked operations**, especially in debug mode, to prevent undefined behavior.

```
char buffer[5] = {'a','b','c','d','e'};
std::string s(buffer, 5); // safe
// std::string s(buffer, 10); // compile-time or runtime diagnostic in debug
↪  mode
```

2. **Disallow Dangling References**

- `std::string_view` cannot implicitly extend the lifetime of temporary strings.
- Explicit constructions encourage **correct usage**:

```
std::string_view sv("hello"); // points to string literal, safe
std::string_view sv2(std::string("hello")); // unsafe: temporary destroyed
↪  immediately
```

- The compiler may warn about **temporary lifetime issues**.

3. **Iterator-Based Construction**

- When constructing strings from **iterators**, C++23 ensures iterators are **checked for validity** in debug builds.
- Supports **safe use of input, forward, and random-access iterators**:

```
std::vector<char> v{'a','b','c'};
std::string s(v.begin(), v.end()); // safe
```

20.3.3 `resize_and_overwrite` Integration

C++23 encourages **in-place construction of string content** to prevent unnecessary copies or invalid access:

```
std::string s;
s.resize_and_overwrite(5, [](char* buf, size_t) -> size_t {
    std::copy_n("Hello", 5, buf);
    return 5; // safe length returned
});
```

- Guarantees **correct string size** after construction.
- Eliminates risks of **buffer overrun**.

20.3.4 Safer `substr` and `assign`

- `substr` now **returns an empty string** if requested position exceeds the original string size.
- `assign` methods now **enforce safe bounds** when copying from pointers, iterators, or string_views:

```
std::string s = "C++23";
auto sub = s.substr(10, 5); // returns empty string, safe
```

- Prevents common **out-of-bounds bugs**.

20.3.5 Improved `std::string_view` Lifetime Management

- C++23 clarifies that `std::string_view` **does not own memory**.
- Safer constructions ensure **programmers are aware** of lifetime issues, especially when pointing to temporaries:

```
std::string_view sv;
{
    std::string temp = "world";
    sv = temp; // discouraged; lifetime ends with temp
}
```

- Compiler diagnostics may help prevent dangling views.

20.3.6 Benefits

Safety Improvement	Description
Bounds checking	Prevents access beyond allocated memory during construction.
Dangling pointer awareness	Ensures `string_view` does not refer to destroyed temporaries.
Iterator validation	Guarantees iterators used in construction are valid.
Integration with `resize_and_overwrite`	Enables safe in-place string construction.

Safety Improvement	Description
Safe `substr` and `assign`	Avoids exceptions or undefined behavior from invalid positions.

Overall Advantages:

- Reduces **runtime errors and undefined behavior** in string operations.
- Improves **program safety** without sacrificing performance.
- Encourages **modern, clean, and expressive string handling** practices.

20.3.7 Example: Safer String Construction

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

int main() {
    // Safe construction from literal
    std::string_view sv1("Hello");

    // Safe construction from iterator
    std::vector<char> v{'C','+','+','2','3'};
    std::string s(v.begin(), v.end());

    // Safe substr
    std::string sub = s.substr(10, 5); // returns empty string

    std::cout << sv1 << " " << s << " " << sub << "\n"; // Output: Hello C++23
}
```

This example demonstrates **safe creation, substring extraction, and string view handling** in line with C++23 rules.

20.3.8 Summary

C++23 string construction rules focus on **safety and correctness**:

1. **Bounds enforcement** for pointer and iterator-based construction.
2. **Lifetime awareness** for `string_view` to prevent dangling references.
3. **Safe substring and assignment operations**.
4. **Integration with in-place construction utilities** like `resize_and_overwrite`.

These rules **reduce common string bugs** while maintaining **performance and flexibility**, helping developers write **more robust and maintainable code** in modern C++.

Chapter 21

I/O and Printing

21.1 `std::print` and `std::println`

C++23 introduces **two new high-level output functions** in `<iostream>` and `<format>`:

- **`std::print`** – prints formatted text to standard output.
- **`std::println`** – prints formatted text followed by a newline character.

These functions **simplify text output**, making it easier and safer to print formatted content compared to the traditional combination of `std::cout` and `operator<<`. They leverage the **`std::format` style syntax**, enabling **modern formatting capabilities** in a concise manner.

21.1.1 Motivation

Before C++23, printing required verbose code for formatted output:

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

int main() {
    int x = 42;
    std::cout << "Value: " << x << "\n";
}
```

- Manual concatenation or multiple `operator<<` calls were needed.
- Format specifiers were limited to `printf`-style functions, which lacked **type safety**.

C++23 introduces `std::print` and `std::println` to provide:

- **Type-safe formatted output**
- **Concise syntax** similar to `printf` but safer
- **Automatic newline handling** with `std::println`

21.1.2 `std::print`

Overview `std::print` is a **variadic template function** that prints **formatted content** to the standard output (`stdout`).

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

int main() {
    int x = 42;
    std::print("Value: {}\n", x); // prints: Value: 42
}
```

Key Features:

- Accepts **format string** with `{}` placeholders.
- Supports **positional arguments** and **format specifiers**.
- Can handle **built-in types, strings, ranges, tuples, and user-defined types with custom formatters**.

```
#include <format>
#include <vector>

int main() {
    std::vector<int> v{1,2,3};
    std::print("Vector contents: {}\n", v); // prints: Vector contents: [1, 2, 3]
}
```

Example
Advantages:

- Eliminates verbose `operator<<` chains.
- Type-safe: mismatched types in format placeholders produce **compile-time errors**.
- Integrates with **C++23 range and tuple formatting**.

21.1.3 `std::println`

Overview `std::println` is similar to `std::print`, but **automatically appends a newline** (`\n`) at the end of the output:

```
#include <format>

int main() {
    int x = 42;
    std::println("Value: {}", x); // Output: Value: 42\n
}
```

Key Features:

- Reduces the risk of forgetting newline characters.
- Fully supports **format specifiers** and **range/tuple formatting**.
- Can be combined with **user-defined formatters** for structured types.

```
#include <format>
#include <tuple>

int main() {
    auto t = std::make_tuple("C++23", 2023);
    std::println("Tuple: {}", t); // Output: Tuple: ("C++23", 2023)
}
```

Example

Advantages:

- Enhances **readability and conciseness** in console applications.
- Works consistently with **all standard C++23 formatting features**.

21.1.4 Customization and Flexibility

C++23 allows `std::print` and `std::println` to be **redirected to other output streams** using `std::print(std::FILE* stream, ...)`.

```
#include <cstdio>
#include <format>

int main() {
    std::print(stderr, "Error: {}\n", "File not found"); // prints to stderr
}
```

- Supports **standard streams** (`stdout`, `stderr`) or **file streams**.
- Provides **consistent formatting semantics** across different output targets.

21.1.5 Combined Example

```
#include <format>
#include <vector>
#include <tuple>

int main() {
    int a = 10;
    double b = 3.1415;
    std::vector<int> v{1,2,3};
    auto t = std::make_tuple(a, b);

    std::println("Integer: {}, Double: {:.2f}", a, b);
    std::println("Vector: {}", v);
```

```
    std::println("Tuple: {}", t);
}
```

Output:

```
Integer: 10, Double: 3.14
Vector: [1, 2, 3]
Tuple: (10, 3.1415)
```

- Demonstrates **type-safe, readable, and concise output** for diverse data types.

21.1.6 Benefits of `std::print` and `std::println`

Feature	Advantage
Type-safe formatting	Compile-time checking of types and format specifiers.
Variadic arguments	Flexible formatting for multiple values in a single call.
Automatic newline	`std::println` appends newline automatically.
Range and tuple support	Direct printing of containers and heterogeneous tuples.
Stream redirection	Output to `stdout`, `stderr`, or custom streams.

Overall Advantages:

- Simplifies **console I/O in modern C++**
- Reduces **boilerplate code** for formatted printing
- Encourages **safe and expressive output patterns**

- Integrates seamlessly with **other C++23 library improvements** such as ranges, tuples, and string formatting

These additions mark a significant improvement in **text output capabilities** for C++23, making **printing concise, type-safe, and flexible**, while supporting modern formatting paradigms.

21.2 Spanstream Library

C++23 introduces **`<spanstream>`**, a new standard library header that provides **`std::ispanstream` and `std::ospanstream`**, enabling **stream-based I/O operations directly on `std::span` objects**.
This addition significantly improves **memory-safe, efficient in-memory I/O**, particularly for temporary buffers, arrays, and other contiguous memory structures without requiring dynamic memory allocation.

21.2.1 Motivation

Prior to C++23, performing stream-like operations on raw buffers or fixed-size arrays required **`std::stringstream`** combined with copies or manual management:

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

int main() {
    std::array<char, 10> buffer;
    std::stringstream ss;
    ss << 123;
```

```
    ss.read(buffer.data(), buffer.size());
}
```

Limitations:

- Extra **heap allocation** in `std::stringstream`.
- Manual **buffer management** and size checking.
- No direct support for **span-based memory buffers**.

C++23 solves these issues by allowing **streams to operate directly on spans**, providing **safer, allocation-free, in-place I/O operations**.

21.2.2 Key Components

C++23 `<spanstream>` defines two main classes:

1. **`std::ispanstream`**
    - Input stream that reads from a **`std::span<const char>`** or other byte-like spans.
    - Behaves like `std::istream`, supporting **formatted and unformatted input**.

```
#include <spanstream>
#include <span>
#include <iostream>

int main() {
    char buf[] = "42 3.14 hello";
```

```
    std::ispanstream ss(buf);

    int i;
    double d;
    std::string str;

    ss >> i >> d >> str;
    std::cout << i << ", " << d << ", " << str << "\n"; // Output: 42, 3.14,
    ↪  hello
}
```

Features:

- Reads **directly from contiguous memory** without copying.
- Supports **all standard stream extraction operators** (>>).
- **Bounds-safe**: cannot read beyond the span size.

2. **`std::ospanstream`**

- Output stream that writes into a **`std::span<char>`** or other writable contiguous buffers.
- Supports **formatted output** without dynamic allocation.

```
#include <spanstream>
#include <span>
#include <iostream>

int main() {
    char buffer[20];
```

```
    std::ospanstream ss(buffer);

    int i = 100;
    double d = 3.14;
    ss << i << " " << d;

    std::cout << std::string_view(buffer, ss.view().size()) << "\n"; // Output:
    ↪  100 3.14
}
```

Features:

- **Writes in-place** to spans, avoiding heap allocation.
- Provides `.view()` to get a `std::span<const char>` of written data.
- Compatible with **format specifiers and manipulators** like `std::setw` or `std::setprecision`.

21.2.3 Benefits of Spanstream

Feature	Advantage
Allocation-free	Operates on existing buffers without heap allocation.
Bounds-safe	Reads/writes are limited to the span size.
Stream compatibility	Supports standard `operator<<` and `operator>>` for types.
Zero-copy operations	No temporary string or buffer objects needed.

Feature	Advantage
Works with `std::span`	Seamlessly integrates with C++20/23 span-based APIs.

Use Cases:

- Parsing or formatting **fixed-size buffers** in embedded systems.
- Efficient **serialization/deserialization** in memory-sensitive applications.
- Temporary **in-memory string manipulation** without allocating dynamic memory.

21.2.4 Combined Example: Input and Output

```
#include <spanstream>
#include <span>
#include <iostream>

int main() {
    char input_buffer[] = "42 3.14 example";
    std::ispanstream in(input_buffer);

    int x;
    double y;
    std::string z;
    in >> x >> y >> z;

    char output_buffer[50];
    std::ospanstream out(output_buffer);
```

```
    out << "Read values: " << x << ", " << y << ", " << z;

    std::cout << std::string_view(output_buffer, out.view().size()) << "\n";
    // Output: Read values: 42, 3.14, example
}
```

- Demonstrates **in-place reading and writing** with spans.
- No dynamic allocation or copying required.
- Fully type-safe and bounds-checked.

21.2.5 Advantages Over Traditional String Streams

Aspect	std::stringstream	std::spanstream
Allocation	Uses dynamic memory	Uses existing buffer/span
Safety	Manual buffer checks	Bounds-safe by design
Performance	May involve copies	Zero-copy, in-place
Compatibility	Standard stream interface	Standard stream operators supported
Use in Embedded Systems	Less efficient	Highly efficient

Conclusion:

The `<spanstream>` library in C++23 provides **safe, efficient, and allocation-free stream operations** on contiguous memory. It modernizes **in-memory I/O**,

reduces boilerplate code, and is ideal for **high-performance and memory-sensitive applications**.

21.3 Enhancements to `fstream` and `ostream`

C++23 introduces several **significant enhancements** to **file and output stream classes**, particularly `std::fstream`, `std::ifstream`, `std::ofstream`, and `std::ostream`. These changes focus on **performance, flexibility, and modern usability**, addressing long-standing limitations in the I/O library.

21.3.1 Motivation

Prior to C++23, stream classes had limitations:

- **Manual buffer management** was required for performance-critical file operations.
- `std::ostream` lacked direct **formatting integration** for modern types like **ranges, tuples, and `std::string_view`**.
- File I/O sometimes required verbose code to manage opening modes, error states, or stream positioning.

C++23 addresses these issues by **streamlining file operations, improving type support, and providing better safety and diagnostics**.

21.3.2 `fstream` Enhancements

1. **Direct `std::format` Integration**

C++23 allows writing formatted content directly to **file streams** using **`std::format`-style syntax**:

```
#include <fstream>
#include <format>

int main() {
    std::ofstream file("output.txt");

    int a = 42;
    double b = 3.1415;

    file << std::format("Integer: {}, Double: {:.2f}\n", a, b);
}
```

Benefits:

- Eliminates the need for intermediate `std::stringstream`.
- Type-safe formatting ensures **compile-time checks**.
- Supports **ranges and tuples**, consistent with `std::print`.

2. **Safer File Opening and Error Handling**

   - `std::fstream` now supports **safer constructors** that can detect errors immediately:

```
std::ofstream out("file.txt", std::ios::out | std::ios::trunc);
if (!out) {
    throw std::runtime_error("Failed to open file");
}
```

- Streams now provide **clearer diagnostics** for failures, reducing silent errors.

3. **Performance Improvements**

   - Improved **buffering strategies** for file streams.
   - Optimized **large file I/O** and **aligned memory writes**.
   - Reduced unnecessary temporary allocations during formatted writes.

21.3.3 `ostream` Enhancements

C++23 extends **`std::ostream` capabilities** to support:

1. **Direct Output of Ranges and Tuples**

   - Previously, printing a `std::vector` or `std::tuple` required manual loops.
   - Now, `ostream` integrates with **C++23 formatting library**, allowing direct output:

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

int main() {
    std::vector<int> v{1, 2, 3};
    std::tuple<int, double, const char*> t{42, 3.14, "C++23"};

    std::cout << v << "\n"; // Output: [1, 2, 3]
    std::cout << t << "\n"; // Output: (42, 3.14, "C++23")
}
```

2. **Safer `operator<<` for `std::string_view`**

   - `std::ostream` now handles `string_view` safely, **avoiding undefined behavior** when printing temporary objects.

```
#include <iostream>
#include <string_view>

std::cout << std::string_view("Hello, C++23!") << "\n";
```

   - Ensures the output remains valid, even with **short-lived string_views**.

3. **Custom Formatter Integration**

   - Users can define **custom `operator<<` or `std::formatter`** for user-defined types.
   - This allows **seamless output of complex types**, including ranges and nested containers.

21.3.4 Combined Example: File and Console Output

```
#include <fstream>
#include <iostream>
#include <vector>
#include <tuple>
#include <format>

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

```
    auto t = std::make_tuple("C++23", 2023);

    std::ofstream file("output.txt");
    file << std::format("Vector: {}\nTuple: {}\n", v, t);

    std::cout << "Written to file successfully.\n";
}
```

Output in `output.txt`:

```
Vector: [10, 20, 30]
Tuple: ("C++23", 2023)
```

- Demonstrates **type-safe, formatted output** directly to files.
- No need for temporary strings or streams.

21.3.5 Benefits

Feature	Advantage
Direct `std::format` in streams	Eliminates intermediate string streams, improves type safety
Safer file opening	Immediate error detection, robust exception handling
Ranges and tuple output	Uniform, concise, and readable output for modern container types
Optimized buffering	Improved performance for large file I/O

Feature	Advantage
Safe `string_view` handling	Prevents dangling references and undefined behavior
Custom formatter support	Allows user-defined types to integrate seamlessly with streams

Overall Advantages:

- Modernizes **file and console output** in C++23.
- Reduces **boilerplate code** and improves **safety and readability**.
- Integrates with **other C++23 library features**, including ranges, tuples, strings, and formatting utilities.

21.3.6 Summary

C++23 enhancements to `fstream` and `ostream` provide:

1. **Formatted output with `std::format`**, including ranges and tuples.
2. **Safer file opening and error diagnostics**.
3. **Optimized in-memory buffering** for performance-critical operations.
4. **Improved type safety** for `string_view` and user-defined types.
5. **Seamless integration** with modern C++23 I/O features such as `std::print` and `std::spanstream`.

These updates make **C++23 I/O more robust, concise, and expressive**, providing a modern alternative to legacy `iostream` patterns.

Part IV

Removed, Deprecated, and Reverted Features

Chapter 22

Removed Features

22.1 Garbage Collection Support

C++23 officially **removes the optional garbage collection (GC) support** that existed in prior standards, such as hooks and facilities related to automatic memory management through garbage collectors. This change reflects the continued **philosophy of C++ as a deterministic and resource-aware language**, emphasizing **manual memory management, smart pointers, and RAII (Resource Acquisition Is Initialization)**.

22.1.1 Background

C++ originally included **optional garbage collection hooks**, primarily through:

- `std::declare_reachable`, `std::undeclare_reachable`
- `std::collectible`
- Functions supporting GC-aware allocation for experimental implementations

These were intended to allow:

- Integration with **conservative or precise garbage collectors**
- Automatic reclamation of dynamically allocated memory without explicit `delete`
- Easier memory management for certain application domains

However, these features were **rarely implemented or used** in production code due to:

1. **Complexity and limited adoption** – few compilers or libraries supported GC hooks.
2. **Non-deterministic performance** – GC introduces unpredictability in execution time.
3. **Conflict with RAII** – modern C++ idioms favor **deterministic destruction**, using smart pointers and scoped resources.

As a result, the C++ standards committee decided to **remove garbage collection support in C++23**.

22.1.2 Impact of Removal

With C++23:

- **All GC-related functions, classes, and headers** are officially removed.
- Attempting to use features like `std::declare_reachable` or `std::collectible` results in **compile-time errors**.
- Libraries and compilers that previously offered optional GC support are **no longer standardized**.

22.1.3 Recommended Alternatives

C++23 encourages developers to rely on **modern, standard mechanisms for memory management**:

1. **Smart Pointers**
   - `std::unique_ptr` – sole ownership with deterministic destruction
   - `std::shared_ptr` – reference-counted shared ownership
   - `std::weak_ptr` – non-owning reference to avoid cyclic references

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

struct Node {
    int value;
};

int main() {
    std::unique_ptr<Node> node = std::make_unique<Node>();
    node->value = 42;
    std::cout << node->value << "\n"; // 42
} // node automatically destroyed here
```

2. **Standard Containers**
   - `std::vector`, `std::map`, and other standard containers **manage memory internally**, reducing the need for manual allocation.

3. **RAII Patterns**

- Deterministic resource management ensures **automatic cleanup** of memory and other resources (files, sockets, locks) when objects go out of scope.

```
#include <fstream>
void writeFile() {
    std::ofstream file("data.txt"); // RAII ensures file is closed on scope
    ↪  exit
}
```

4. **Manual Memory Management**

- Direct `new` and `delete` remain fully supported.
- Deterministic control is preferred for **performance-critical systems**.

22.1.4 Advantages of Removal

Removed Feature	Advantage
Garbage collection	Reduces **complexity** and standard overhead
GC hooks & support	Encourages use of **RAII and smart pointers**
Non-deterministic GC	Ensures **predictable performance** in all environments
Rarely adopted APIs	Simplifies the standard and focuses on widely used features

Overall, removal aligns C++23 with the language's **core philosophy of resource determinism and high performance**.

22.1.5 Example: Modern Memory Management

Instead of relying on garbage collection:

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

struct Data {
    int x;
};

int main() {
    std::vector<std::unique_ptr<Data>> container;

    for (int i = 0; i < 5; ++i) {
        container.push_back(std::make_unique<Data>());
        container.back()->x = i * 10;
    }

    for (auto& item : container) {
        std::cout << item->x << " ";
    }
    // Output: 0 10 20 30 40
}
```

- **Automatic cleanup** occurs when `container` goes out of scope.
- No need for a garbage collector.
- Memory management is **explicit, safe, and efficient**.

22.1.6 Summary

C++23 removes **all optional garbage collection support**, emphasizing:

1. **Deterministic memory management** through RAII.
2. **Smart pointer usage** for safe ownership semantics.
3. **Avoidance of non-deterministic and rarely implemented features**.
4. **Simplification of the standard library** by focusing on widely adopted and modern memory management idioms.

Developers are encouraged to **fully adopt smart pointers, RAII, and standard containers** for memory safety, performance, and maintainability in modern C++ programming.

22.2 Mixed Wide String Concatenation

C++23 removes support for **implicit concatenation of narrow and wide string literals**. This change reflects the standard's ongoing effort to **eliminate ambiguous or unsafe behaviors** in string handling and to **encourage explicit type conversions** for clarity and correctness.

22.2.1 Background

Prior to C++23, C++ allowed code such as:

```
#include <iostream>

int main() {
    auto s = L"Hello " "World"; // wide + narrow literal concatenation
    std::wcout << s << "\n";
}
```

- Here, a **wide string literal (`L"Hello "`)** was concatenated with a **narrow string literal (`"World"`)**.
- The compiler would implicitly convert the narrow literal to a wide string.

Problems with this approach:

1. **Type ambiguity** – automatic conversion between `char` and `wchar_t` can cause unexpected behavior.
2. **Portability issues** – behavior varied across compilers and platforms.
3. **Maintenance risks** – mixing literals of different widths can introduce subtle bugs in string operations, formatting, and I/O.

Due to these issues, C++23 **removes this implicit support**, enforcing **strict type matching** in string concatenation.

22.2.2 New Rules

C++23 requires that:

- **All string literals in a concatenation must have the same type** (`char`, `wchar_t`, `char8_t`, `char16_t`, or `char32_t`).
- Implicit conversion between literal types is no longer performed by the compiler.

```
#include <iostream>

int main() {
    // std::wstring s = L"Hello " "World"; // Error in C++23

    std::wstring s = L"Hello " L"World"; // Correct: both wide literals
    std::wcout << s << "\n";
}
```

- Using **mixed literals** now results in a **compile-time error**, enforcing type safety.
- Encourages developers to **explicitly convert literals** if mixing is necessary.

22.2.3 Recommended Alternatives

1. **Use Consistent Literal Types**

   Always ensure all literals in a concatenation are of the same type:

```
// Wide string example
std::wstring s1 = L"Wide " L"String";

// Narrow string example
std::string s2 = "Narrow " "String";
```

- Improves **readability, maintainability, and portability**.

2. **Explicit Conversion**

   If mixing literal types is necessary, use **explicit conversion functions**:

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

int main() {
    const char* narrow = "World";
    std::wstring wide = L"Hello " + std::wstring(narrow, narrow +
    ↪  std::strlen(narrow));
    std::wcout << wide << "\n";
}
```

- Converts the narrow string to a wide string explicitly.
- Eliminates ambiguity and compiler-dependent behavior.

3. **Use `std::format` or `std::print`**

   C++23's **formatting utilities** simplify mixed-width output:

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

int main() {
    std::wcout << std::format(L"{} {}", L"Hello", L"World") << "\n";
}
```

- Provides **type-safe, readable concatenation**.
- Works with **strings, string_views, and ranges**.

22.2.4 Advantages of Removal

Removed Feature	Advantage
Mixed narrow/wide concatenation	Eliminates ambiguous behavior
Implicit conversions between `char` and `wchar_t`	Prevents subtle bugs and improves portability
Compiler-dependent behavior	Ensures consistent behavior across platforms
Unsafe concatenation practices	Encourages explicit, readable, and maintainable string handling

Overall, this removal strengthens **type safety** and aligns string handling with modern C++ best practices.

22.2.5 Example: Correct and Incorrect Usage

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

int main() {
    // Incorrect in C++23:
    // std::wstring s = L"Hello " "World"; // compile-time error

    // Correct usage:
    std::wstring s1 = L"Hello " L"World"; // wide string literals
    std::string s2 = "Hello " "World";    // narrow string literals

    std::wcout << s1 << "\n";
    std::cout << s2 << "\n";
}
```

Output:

```
Hello World
Hello World
```

- Demonstrates **strict type consistency** for literals in C++23.
- Encourages **modern, safe string concatenation patterns**.

22.2.6 Summary

C++23 removes **mixed narrow and wide string literal concatenation**, enforcing:

1. **Strict type consistency** for all string literals.

2. **Compile-time errors** when implicit conversions are attempted.

3. Encouragement to use **explicit conversions, smart string handling, or modern formatting utilities**.

This change reduces **bugs, ambiguity, and portability issues**, aligning with **C++23's philosophy of safety, clarity, and maintainability** in text processing.

22.3 Non-Encodable Wide Characters

C++23 removes support for **non-encodable wide characters**—those `wchar_t` or `char16_t/char32_t` code units that cannot be represented in the execution character set. This decision reflects the standard's goal of **ensuring well-defined behavior and eliminating undefined or implementation-defined cases** in string handling and character encoding.

22.3.1 Background

In earlier C++ standards, it was possible to declare **wide characters that did not correspond to valid encodings in the target execution character set**, for example:

```
wchar_t wc = L'\uDC00'; // surrogate code unit
```

- `L'\uDC00'` is a **UTF-16 low surrogate**, which is **non-encodable as a standalone character**.
- Operations involving such characters, including **streaming, concatenation, or comparison**, had **undefined or implementation-defined behavior**.

Problems caused by non-encodable wide characters included:

1. **Undefined behavior in I/O operations** (writing to `std::wostream`).
2. **Incorrect string manipulations**, like length calculation or substring extraction.
3. **Portability issues** across different compilers and platforms.

Due to these problems, C++23 **removes support for non-encodable wide characters**, requiring that all wide character literals be **representable in the execution character set**.

22.3.2 New Rules in C++23

1. **Wide character literals** must correspond to **encodable values** in the current execution character set.
2. Characters that **cannot be encoded** now result in **compile-time errors**.
3. Any **operations on previously non-encodable characters** are disallowed, ensuring well-defined behavior.

```
// Pre-C++23 (potentially UB)
wchar_t wc = L'\uDC00'; // may compile, behavior undefined

// C++23 (compile-time error)
wchar_t wc = L'\uDC00'; // error: non-encodable character
```

- Ensures **safety and consistency** when working with `wchar_t`, `char16_t`, and `char32_t`.
- Improves **portability** for cross-platform text processing.

22.3.3 Recommended Alternatives

1. **Use Unicode Code Points Properly**
    - Ensure all characters used in **wide string literals** are **valid code points**:

```
wchar_t valid_char = L'\u263A'; //  , valid Unicode code point
```

- Avoid using **surrogate halves** (`\uD800-\uDFFF`) directly.

2. **Use UTF Encodings with Strings**

- Prefer **UTF-8, UTF-16, or UTF-32 strings** (`std::u8string`, `std::u16string`, `std::u32string`) and **encoding libraries**:

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

std::u16string utf16 = u"Hello \u263A";
std::cout << "UTF-16 length: " << utf16.size() << "\n";
```

- Libraries like **ICU** or **std::format + std::u8string** help safely handle Unicode text.

3. **Avoid Surrogate Handling Manually**

- Do not manipulate **UTF-16 surrogate halves** directly.
- Use **Unicode-aware functions** for splitting, concatenation, and validation.

22.3.4 Advantages of Removal

Removed Feature	Advantage
Non-encodable wide characters	Eliminates undefined behavior in string and character operations
Compiler-dependent handling	Ensures consistent, portable code across all platforms
Unsafe manipulations	Encourages use of proper Unicode and standard encoding libraries
Legacy text handling quirks	Simplifies modern C++ text processing and I/O

Overall, removing non-encodable wide characters strengthens **type safety, encoding correctness, and portability** in C++23.

22.3.5 Example: Correct Wide Character Usage

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

int main() {
    // Correct wide string literal
    std::wstring ws = L"Hello  "; //   is encodable in wchar_t on most platforms

    for (wchar_t c : ws) {
        std::wcout << c << L' ';
    }
    std::wcout << "\n"; // Output: H e l l o
}
```

- Demonstrates **safe and well-defined iteration** over wide string characters.

- Eliminates risks associated with **non-encodable or surrogate characters**.

22.3.6 Summary

C++23 removes **support for non-encodable wide characters** to ensure:

1. **All wide characters are representable** in the execution character set.
2. **Compile-time detection** of invalid characters.
3. **Well-defined behavior** in I/O, concatenation, and string operations.
4. **Encouragement to adopt Unicode-aware types** and libraries for modern text processing.

This change contributes to **safer, portable, and predictable string handling** in modern C++ programming.

Chapter 23

Deprecated Features

23.1 `std::aligned_storage`, `std::aligned_union`, `numeric_limits::has_denorm`

C++23 continues the trend of **modernizing the standard library** by **deprecating certain older features** that are either superseded by better alternatives or rarely used in practice. Among these, `std::aligned_storage`, `std::aligned_union`, and `numeric_limits::has_denorm` have been officially **deprecated**, signaling developers to transition to safer and more robust constructs.

23.1.1 Background

1. **`std::aligned_storage`**

    - Provided a **compile-time mechanism** to create uninitialized storage suitable for any type with a given size and alignment.

- Typical usage involved **manual memory management** and **placement new**:

```
#include <type_traits>
#include <new>

struct MyType { int x; double y; };

std::aligned_storage<sizeof(MyType), alignof(MyType)>::type storage;
MyType* ptr = new(&storage) MyType{42, 3.14};
```

- Challenges:
  - Verbose syntax.
  - Manual object construction/destruction.
  - Error-prone, especially for complex or non-trivial types.

Replacement: `std::byte` arrays with `std::bit_cast`, or `std::unique_ptr` with custom deleters. Modern C++ emphasizes **type safety and RAII** rather than raw aligned storage.

2. **`std::aligned_union`**

- Allowed creating a type **capable of holding any of a set of types**, aligned appropriately.
- Example:

```cpp
#include <type_traits>

using Storage = std::aligned_union<0, int, double>::type;
Storage storage; // can hold int or double
```

- Challenges:
  - Complex syntax and manual object management.
  - Poor integration with modern type-safe containers.
  - Often replaced by `std::variant` or `std::optional` for **safe type-erased storage**.

Replacement:

- `std::variant<Ts...>` for holding one of multiple types safely.
- `std::optional<T>` for conditional storage of a single type.

3. **`numeric_limits::has_denorm`**

- Provided information on whether a floating-point type supports **denormalized (subnormal) numbers**.
- Example:

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

std::cout << std::numeric_limits<float>::has_denorm << "\n";
```

- Challenges:
  - Often **implementation-dependent**.
  - Rarely used in high-level applications.
  - Modern code typically checks **denorm behavior through constexpr or type traits**.

Replacement:

- `std::numeric_limits<float>::has_denorm_loss` or querying **IEEE 754 compliance** through modern numeric utilities.

23.1.2 Advantages of Deprecation

Deprecated Feature	Reason for Deprecation
`std::aligned_storage`	Encourages unsafe manual memory management; modern alternatives provide safer RAII
`std::aligned_union`	Verbose, error-prone; `std::variant` is type-safe and easier to use
`numeric_limits::has_denorm`	Rarely needed; implementation-dependent and often replaced with modern numeric traits

Overall:

- Deprecation directs developers toward **safer, type-aware, and modern C++ constructs**.
- Helps reduce **manual errors, undefined behavior, and verbose boilerplate**.

23.1.3 Modern Alternatives

1. **Replacing `std::aligned_storage`**

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

struct MyType { int x; double y; };

alignas(MyType) std::byte storage[sizeof(MyType)];
MyType* ptr = std::bit_cast<MyType*>(storage);
new(ptr) MyType{42, 3.14}; // placement new
```

   - Alternatively, use `std::unique_ptr<MyType>` for automatic cleanup.

2. **Replacing `std::aligned_union`**

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

std::variant<int, double> v;
v = 3.14; // safe type holding double
v = 42;   // now holds int
```

   - Provides **type safety** and **automatic destruction** of contained object.

3. **Replacing `numeric_limits::has_denorm`**

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

bool supports_denorm = std::numeric_limits<float>::has_denorm !=
↪  std::denorm_absent;
std::cout << std::boolalpha << supports_denorm << "\n";
```

- Provides **portable detection** of denormal support without relying on deprecated constants.

23.1.4 Summary

C++23 **deprecates `std::aligned_storage`, `std::aligned_union`, and `numeric_limits::has_denorm`** to:

1. Encourage **safer memory management** using RAII and smart pointers.
2. Promote **type-safe alternatives** like `std::variant` and `std::optional`.
3. Replace implementation-dependent floating-point traits with **modern, portable checks**.
4. Reduce **complex, error-prone boilerplate** in modern C++ code.

By adopting the modern replacements, developers achieve **better safety, maintainability, and clarity** in C++23 applications.

Chapter 24

Reverted Deprecations

24.1 Subscript Operator ,

C++23 introduces a **reversion of a previously deprecated feature** related to the **comma operator in subscript expressions**. This section examines the history, rationale, and practical implications of this reversion, clarifying how developers can safely and correctly use the subscript operator with comma expressions.

24.1.1 Background

In earlier C++ standards prior to C++17:

- It was possible to write subscript expressions using the **comma operator** inside the index:

```
int arr[5] = {10, 20, 30, 40, 50};
int value = arr[1, 3]; // ambiguous expression
```

- The expression `1, 3` is a **comma operator**, which evaluates its left operand (`1`), discards the result, and then evaluates the right operand (`3`).
- Consequently, `arr[1, 3]` **accesses** `arr[3]`, not `arr[1]`.

While legal, this usage:

1. **Caused confusion** for many programmers, as the behavior is **not immediately obvious**.
2. **Increased potential for subtle bugs**, particularly in complex expressions.
3. Was considered **unintuitive** for newcomers to C++.

Due to these concerns, **the standard committee deprecated such usage in C++17**.

24.1.2 Reversion in C++23

C++23 **reverts the deprecation**, meaning:

- **Using the comma operator inside subscript expressions is no longer deprecated.**
- The behavior remains **well-defined and consistent** with the standard semantics of the comma operator.
- This change reflects the philosophy of **maintaining backward compatibility** while keeping the language flexible for experienced developers.

```
int arr[5] = {10, 20, 30, 40, 50};
int value = arr[1, 3]; // Legal in C++23, value == 40
```

- The left operand (`1`) is evaluated and discarded.
- The right operand (`3`) determines the index into the array.

24.1.3 Rationale for Reversion

1. **Consistency with existing language rules** – The comma operator has always been defined to evaluate both operands in order, discarding the left result. Subscript expressions follow normal operator precedence.
2. **Minimal risk to experienced developers** – While potentially confusing for beginners, this construct is **rarely used in complex code** and has predictable behavior.
3. **Encouragement of explicit coding practices** – Developers can still write `arr[(1,3)]` or split expressions for clarity if needed.

24.1.4 Recommended Practices

Even though the comma operator is now legal in subscript expressions, **best practices suggest**:

1. **Use Clear Indexing**
    - Prefer explicit index values for readability:

```
v
```

2. **Use Parentheses for Complex Expressions**

   - If a comma expression is unavoidable, enclose it in parentheses to **highlight intent**:

```
int i = 1, j = 3;
int value = arr[(i, j)]; // clearly evaluates to j
```

3. **Avoid in Critical Code**

   - Avoid comma expressions in array subscripts in **production-critical code**, especially where clarity and maintainability are priorities.

24.1.5 Advantages of Reversion

Feature	Advantage
Subscript with comma operator	Preserves **backward compatibility**
Evaluates both operands	Consistent with **standard operator semantics**
Flexible usage	Allows advanced metaprogramming and concise expressions where safe
Compiler support	No warnings or deprecated behavior in C++23

Overall, the reversion supports **experienced developers** who rely on legacy idioms or concise expression evaluation while maintaining **well-defined behavior**.

24.1.6 Example Usage

```cpp
#include <iostream>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int a = 1, b = 3;

    int value1 = arr[1, 3];       // 40, comma operator discards 1
    int value2 = arr[(a, b)];     // 40, parentheses clarify intent

    std::cout << value1 << " " << value2 << "\n"; // Output: 40 40
}
```

- Demonstrates that **comma in subscript is fully supported**.
- Parentheses can improve readability without changing semantics.

24.1.7 Summary

C++23 **reverts the previous deprecation of using the comma operator in subscript expressions**:

1. **Behavior is well-defined**, consistent with the comma operator.
2. **Backward-compatible**, preserving legacy code correctness.
3. **Encourages cautious and clear usage**, using parentheses when needed.
4. Maintains flexibility for **advanced expression patterns** while avoiding unnecessary compiler warnings.

This reversion reflects the **balance in modern C++** between **safety, clarity, and backward compatibility**.

24.2 C Headers Kept for Compatibility

C++23 continues to maintain **a set of C standard library headers** for backward compatibility, ensuring that **legacy C-style code can coexist with modern C++ projects**. While some features of the C standard library have evolved, the core headers are **kept intact**, reflecting the standard committee's commitment to **smooth migration and interoperability**.

24.2.1 Background

C++ originally adopted the C standard library as a foundation for low-level operations such as:

- Input/output (`<stdio.h>`)
- Memory management (`<stdlib.h>`)
- String manipulation (`<string.h>`)
- Mathematics (`<math.h>`)

To integrate C headers into C++ safely, the standard introduced **C++-style versions of these headers** with the `c` prefix:

C Header	C++ Header Equivalent
`<stdio.h>`	`<cstdio>`
`<stdlib.h>`	`<cstdlib>`
`<string.h>`	`<cstring>`
`<math.h>`	`<cmath>`

C Header	C++ Header Equivalent
`<time.h>`	`<ctime>`
`<assert.h>`	`<cassert>`

- The `c`-prefixed headers place **all declarations in the `std` namespace**, preventing conflicts with global symbols.
- Despite the existence of modern C++ alternatives (like `std::vector`, `std::string`, `std::chrono`), many projects **still rely on traditional C headers**.

24.2.2 C23 / C++23 Status

C++23 **keeps these headers for compatibility**, with the following clarifications:

1. **Both the C-style and C++-style headers are valid.**

```
#include <stdio.h>  // traditional C header
#include <cstdio>   // C++ header, symbols in std::
```

2. **C++ headers guarantee namespace safety** – functions like `printf` and `malloc` exist in the `std` namespace in addition to the global namespace if the C header is included.
3. **No deprecation or removal** – all major C headers are preserved to prevent breaking legacy applications.
4. **Encouragement to prefer C++ alternatives** – modern types like `std::string`, `std::vector`, and `std::chrono` provide safer, type-aware replacements.

24.2.3 Practical Usage

1. **Including C Headers in Modern C++**

```cpp
#include <cstdio>   // C++ style
#include <cstdlib>

int main() {
    std::printf("Hello, World!\n");  // safe in std namespace
    int* arr = static_cast<int*>(std::malloc(5 * sizeof(int)));  // safe
    ↪  allocation
    std::free(arr);
}
```

   - Provides **full compatibility** with C functions.
   - Modern `std::` namespace ensures **no accidental clashes** with custom symbols.

2. **When to Use Traditional Headers**

   - Legacy projects or **ported C code** may include `<stdio.h>` or `<stdlib.h>` directly.
   - C++23 fully supports this usage, making migration **less error-prone**.

3. **Recommended Modern C++ Approach**

   - Prefer **C++-style headers and RAII**:

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

int main() {
    std::vector<int> arr(5, 0);  // automatic memory management
    std::cout << "Vector initialized with zeros.\n";
}
```

- Safer, avoids manual memory management and potential undefined behavior.

24.2.4 Advantages of Keeping C Headers

Feature	Advantage
C headers preserved	Enables **smooth legacy code integration**
C++-style headers (`c` prefix)	Provides **namespace safety** and reduces symbol conflicts
No deprecation in C++23	Maintains **backward compatibility** for large, existing codebases
Interoperability	Supports **mixed C/C++ projects** without modification

- Ensures **portability across compilers** and **preserves developer investment** in existing code.

24.2.5 Summary

C++23 **reverts any potential removal or deprecation of C standard headers**, emphasizing:

1. **Backward compatibility** with C-style code.
2. **Namespace safety** using C++-style headers (`<cxxx>`).
3. Encouragement to **use modern C++ alternatives** where practical.
4. Maintaining **full interoperability** in projects that mix legacy C and modern C++ code.

By keeping C headers, C++23 supports **both modernization and preservation of legacy code**, providing developers the flexibility to gradually adopt safer and more expressive C++ features without breaking existing applications.

Part V

Practical Applications of C++23

Chapter 25

Migrating from C++20 to C++23

Migrating from C++20 to C++23 is generally straightforward, as C++23 builds upon the features of C++20 while introducing **incremental improvements, new libraries, and modern language utilities**. This chapter provides a structured approach for developers to **assess compatibility, adopt new features, and refactor legacy code** to leverage the full capabilities of C++23.

25.1 Overview of Migration Goals

When upgrading a project from C++20 to C++23, developers should aim to:

1. **Ensure compatibility** – confirm existing C++20 code compiles cleanly under C++23.
2. **Adopt new standard library features** – such as `std::expected`, `std::print`, `std::mdspan`, and enhanced ranges and algorithms.
3. **Remove deprecated constructs** – replace or refactor features deprecated or removed in C++23.

4. **Leverage language tweaks** – including expanded `constexpr` support, improved lambdas, and `move_only_function`.

5. **Enhance safety and maintainability** – adopt modern constructs like `std::variant`, `std::optional`, and span-based APIs for safer code.

25.2 Checking Compiler Support

Before migration:

- Ensure the compiler fully supports **C++23 features**. Most major compilers (GCC, Clang, MSVC) provide **full or near-complete C++23 support**.
- Use flags to enable C++23 mode:

```
# GCC / Clang
g++ -std=c++23 main.cpp -o app

# MSVC
cl /std:c++23 main.cpp
```

- Consider compiler-specific notes for experimental features, e.g., `std::generator` or `std::print`.

25.3 Updating the Standard Library Usage

C++23 introduces **several new standard library headers and utilities**:

1. **New Containers and Utilities**

- **`std::mdspan`**: Multidimensional views of contiguous data.
- **`std::flat_map` and `std::flat_set`**: Memory-efficient sorted associative containers.
- **`std::expected`**: Type-safe error handling without exceptions.
- **`std::move_only_function` and `std::bind_back`**: Safer and flexible function wrappers.

Migration tip:
Replace verbose or error-prone patterns with these utilities to improve clarity and maintainability.

```
#include <expected>

std::expected<int, std::string> divide(int a, int b) {
    if (b == 0) return std::unexpected("Division by zero");
    return a / b;
}
```

2. **Ranges and Algorithms**

- New range adaptors: `zip`, `chunk_by`, `adjacent`, etc.
- Constrained algorithms: `find_last`, `starts_with`, `ends_with`.
- Conversion utilities: `ranges::to` for seamless container transformations.

Migration tip:
Replace manual loops or temporary containers with range adaptors for **more expressive and concise code**.

3. **I/O and Formatting**

- `std::print` and `std::println` provide **streamlined formatted output**.
- `spanstream` allows constructing streams over spans without copying.

Migration tip:
Consider replacing `printf`/`iostream` combinations with `std::print` for **simpler, type-safe output**.

25.4 Addressing Removed or Deprecated Features

During migration, check for **deprecated or removed C++20 features**:

Feature	Action in C++23
`std::aligned_storage`	Replace with `std::byte` arrays or RAII constructs
`std::aligned_union`	Use `std::variant` or `std::optional`
`numeric_limits::has_denorm`	Use `has_denorm_loss` or portable numeric traits
Non-encodable wide chars	Ensure all wide character literals are representable
Mixed wide string concatenation	Refactor using `std::wstring` safe concatenation

- Compilers may generate **warnings or errors** if deprecated constructs are used.
- Migration is a **good opportunity to adopt modern C++23 idioms**.

25.5 Updating Compile-Time and `constexpr` Usage

- C++23 **expands `constexpr` support** to:
  - `std::bitset` operations
  - `std::unique_ptr` and smart pointer operations
  - Mathematical functions in `<cmath>`

Migration tip:
Review functions that could now be **evaluated at compile-time**, improving performance and reducing runtime overhead.

25.6 Memory and Safety Improvements

C++23 introduces new **memory management utilities**:

- `std::out_ptr` / `std::inout_ptr` for safe interop with C APIs.
- `allocate_at_least` and `start_lifetime_as` for precise memory allocation and object lifetime management.

Migration tip:
Refactor raw pointer usage to **smart pointers and these new helpers** to reduce memory leaks and undefined behavior.

25.7 Practical Migration Strategy

1. **Compile with C++23 mode** and fix compiler errors/warnings.

2. **Audit code for deprecated features** and plan replacements.
3. **Introduce new standard library features gradually**, starting with:
    - Error handling (`std::expected`)
    - Ranges and algorithms
    - I/O (`std::print`)
4. **Refactor low-level memory code** using `std::out_ptr` and `allocate_at_least`.
5. **Test extensively** – C++23 ensures compatibility but subtle behavior changes may exist with ranges and `constexpr`.
6. **Document changes** for future maintainers.

25.8 Example: Modernizing a C++20 Function

C++20 version using exceptions and loops:

```
int sum_even(const std::vector<int>& v) {
    int sum = 0;
    for (int n : v) {
        if (n % 2 == 0) sum += n;
    }
    return sum;
}
```

C++23 version using ranges and algorithms:

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

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

- Clearer, more expressive, and **fully utilizes C++23 range improvements**.

25.9 Summary

Migrating from C++20 to C++23 involves:

1. Ensuring **compiler support and backward compatibility**.
2. **Replacing deprecated features** with modern alternatives.
3. **Adopting new library features** for containers, ranges, coroutines, I/O, and memory management.
4. Leveraging **expanded `constexpr` capabilities** and safer pointer utilities.
5. Improving **code clarity, maintainability, and performance** by modernizing loops, algorithms, and error handling.

By following these strategies, developers can **seamlessly migrate to C++23**, while taking full advantage of **modern language features and library enhancements**.

Chapter 26

Modernizing Legacy Code with C++23

Legacy C++ codebases often contain patterns and constructs from older standards, such as raw pointers, manual memory management, non-type-safe containers, and verbose loops. C++23 provides a **range of modern features and library enhancements** that can significantly improve code safety, maintainability, and performance. This chapter explores strategies and practical approaches to **modernizing legacy code using C++23**.

26.1 Identifying Legacy Patterns

Common legacy C++ patterns include:

1. **Raw pointers and manual memory management**

```
int* arr = new int[10];
// ... use arr
delete[] arr;
```

2. **C-style arrays and loops**

```
for (int i = 0; i < 10; ++i) arr[i] = i * 2;
```

3. **Exception-based error handling or error codes**

```
int result = divide(a, b);
if (result == -1) handle_error();
```

4. **Verbose type management with unions or manual alignment**

```
std::aligned_storage<sizeof(MyType), alignof(MyType)>::type storage;
```

5. **Manual string manipulation with `char\*` or `std::sprintf`**

Modernizing these patterns improves **safety, readability, and performance** while reducing **undefined behavior and memory leaks**.

26.2 Modern Memory Management

C++23 introduces utilities that replace legacy raw pointers:

1. **Smart Pointers**

- Use `std::unique_ptr`, `std::shared_ptr` instead of raw `new`/`delete`.

```
#include <memory>

auto arr = std::make_unique<int[]>(10); // automatic cleanup
```

2. **Interfacing with Legacy C APIs**

   - Use `std::out_ptr` and `std::inout_ptr` for safe interop:

```
#include <memory>
#include <memory_resource>

std::unique_ptr<FILE, decltype(&fclose)> file(nullptr, &fclose);
std::fopen_s(&file, "data.txt", "r");  // safe initialization
```

3. **Precise Allocation**

   - `allocate_at_least` and `start_lifetime_as` allow **fine-grained memory control** while avoiding raw pointer pitfalls.

26.3 Replacing Legacy Containers

Legacy code often relies on C-style arrays or `std::vector` for multi-dimensional data. C++23 provides:

1. **`std::mdspan`** – lightweight, multidimensional views over contiguous memory.

```
#include <mdspan>
int data[6] = {1, 2, 3, 4, 5, 6};
std::mdspan<int, std::extents<2,3>> view(data);
```

2. **`std::flat_map` and `std::flat_set`** – memory-efficient associative containers.

Migration tip:

- Replace `std::map` or `std::unordered_map` where data is **mostly read-only** for improved cache efficiency.

26.4 Safer Error Handling

Legacy code often uses error codes or exceptions. C++23 introduces:

- **`std::expected`** – a type-safe alternative for error handling without exceptions.

```
#include <expected>
std::expected<int, std::string> divide(int a, int b) {
    if (b == 0) return std::unexpected("Division by zero");
    return a / b;
}
```

- Improves clarity and avoids **exception overhead** in performance-sensitive code.

26.5 Leveraging Ranges and Algorithms

C++23 provides **powerful range adaptors and algorithms** to modernize loops:

- **Legacy loop:**

```
std::vector<int> v = {1,2,3,4,5};
std::vector<int> squares;
for(auto x : v) {
    if(x % 2 == 0) squares.push_back(x*x);
}
```

- **Modern C++23 version:**

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

auto squares = v
               | std::views::filter([](int n){ return n%2==0; })
               | std::views::transform([](int n){ return n*n; })
               | std::ranges::to<std::vector>();
```

  - **Improves readability**
  - **Reduces boilerplate**
  - **Encourages functional programming style**

26.6 Modern I/O

Legacy code often uses `printf`, `sprintf`, or verbose `iostream` patterns. C++23 introduces:

- **`std::print` and `std::println`** – simplified, type-safe formatted output.
- **`spanstream`** – constructing streams directly over spans.

```
#include <print>

int value = 42;
std::print("The answer is {}\n", value);
```

- Safer and avoids buffer overflows compared to `sprintf`.

26.7 Modern Compile-Time Features

C++23 **extends `constexpr` capabilities** to:

- `std::bitset` operations
- `std::unique_ptr`
- Mathematical and string operations

Migration tip:

- Move computations from runtime to **compile-time** where feasible to improve performance.

26.8 Guidelines for Modernization

1. **Audit Legacy Code** – Identify unsafe patterns and deprecated constructs.
2. **Adopt Modern Containers** – Replace raw arrays with `std::vector`, `std::mdspan`, or flat containers.
3. **Refactor Memory Management** – Replace raw pointers with `unique_ptr` and `out_ptr` for safe interop.
4. **Modernize Loops** – Use ranges, views, and `std::ranges::to`.
5. **Enhance Error Handling** – Replace error codes with `std::expected` or structured exceptions.
6. **Update I/O** – Replace `printf` and `sprintf` with `std::print` and `spanstream`.
7. **Leverage `constexpr`** – Move computations to compile-time wherever possible.
8. **Test Extensively** – Ensure functional equivalence after modernization.

26.9 Example: Legacy Function Modernized

Legacy C++20 style:

```
int sum_even(const int* arr, size_t size) {
    int sum = 0;
    for(size_t i = 0; i < size; ++i) {
        if(arr[i] % 2 == 0) sum += arr[i];
    }
    return sum;
}
```

C++23 modernized version:

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

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

- Cleaner, safer, and leverages **modern algorithms and ranges**.

26.10 Summary

Modernizing legacy code with C++23 improves:

- **Safety** – reduces undefined behavior, memory leaks, and buffer overflows.
- **Maintainability** – clearer, more expressive constructs replace verbose boilerplate.
- **Performance** – improved compile-time evaluation, cache-efficient containers, and range algorithms.
- **Readability** – modern patterns like ranges, `std::print`, and `std::expected` make code easier to understand.

C++23 provides developers with **tools to bring legacy projects up to modern standards**, while maintaining **backward compatibility and incremental migration paths**.

Chapter 27

Writing Safer and Faster Code with New Features

C++23 introduces several language and library features designed to help developers **write code that is both safer and more performant**. This chapter explores strategies for leveraging these features, including modern memory management, enhanced type safety, compile-time computation, optimized containers, and efficient algorithms.

27.1 Safer Code with Modern Utilities

C++23 provides tools that **reduce undefined behavior, improve type safety, and enforce better coding practices**:

1. **Type-Safe Error Handling with `std::expected`**
    - Eliminates the need for error codes and reduces exception misuse.

- Encourages explicit handling of failures.

```
#include <expected>
std::expected<int, std::string> divide(int a, int b) {
    if (b == 0) return std::unexpected("Division by zero");
    return a / b;
}

auto result = divide(10, 2);
if(result) {
    // Safe access
    std::cout << result.value();
} else {
    std::cerr << "Error: " << result.error();
}
```

Benefit: Prevents **silent failures** and enforces **error handling** at compile time.

2. **Safer Memory Interfacing**

- **`std::out_ptr` and `std::inout_ptr`** wrap raw pointers when interacting with legacy APIs.
- **`allocate_at_least`** and **`start_lifetime_as`** provide safer low-level memory allocation.

```
#include <memory>
#include <memory_resource>

std::unique_ptr<int[]> arr;
std::allocate_at_least(arr, 10); // ensures minimum allocation
```

Benefit: Reduces **memory leaks, dangling pointers, and undefined behavior**.

27.2 Faster Code with New Containers

C++23 introduces containers and views designed for **better cache locality and efficient iteration**:

1. **`std::flat_map` and `std::flat_set`**
   - Store elements in contiguous memory.
   - Improves **cache efficiency** over traditional `std::map` or `std::set`.

```
#include <flat_map>
std::flat_map<int, std::string> map = {{1,"one"}, {2,"two"}};
```

Benefit: Faster lookup and iteration in **read-heavy scenarios**.

2. **`std::mdspan`**
   - Lightweight, multidimensional **view over existing data**.
   - Avoids copying and supports **efficient indexing** for multi-dimensional arrays.

```
#include <mdspan>
int data[6] = {1,2,3,4,5,6};
std::mdspan<int, std::extents<2,3>> view(data);
```

Benefit: High-performance access to **matrix or tensor data** without overhead.

27.3 Compile-Time Performance Improvements

C++23 **extends `constexpr`** to more constructs:

- `std::bitset` operations
- `std::unique_ptr` and RAII patterns
- Mathematical functions and string operations

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

constexpr int f = factorial(5); // evaluated at compile-time
```

Benefit: Reduces **runtime computation**, improving performance for constant expressions.

27.4 Faster Algorithms with Ranges

- New **range adaptors** (`zip`, `chunk_by`, `adjacent`) and **constrained algorithms** (`find_last`, `starts_with`) optimize iteration and selection.
- `ranges::to` simplifies **container conversion** efficiently.

```
#include <ranges>
#include <vector>
auto even_squares = std::vector{1,2,3,4,5}
                    | std::views::filter([](int n){ return n % 2 == 0; })
```

```
                    | std::views::transform([](int n){ return n*n; })
                    | std::ranges::to<std::vector>();
```

Benefit: Eliminates unnecessary loops and temporary containers, improving both speed and readability.

27.5 Safer String and Text Handling

C++23 enhances **string safety and formatting**:

- `std::string` and `std::string_view` improvements (`contains`, `resize_and_overwrite`, safer construction).
- `std::print` and `std::println` provide **type-safe, efficient formatted output**.
- `spanstream` avoids copying when writing to memory spans.

```
#include <print>
std::string s = "Hello, C++23";
std::print("Message: {}\n", s); // type-safe and concise
```

Benefit: Avoids **buffer overflows** and improves I/O efficiency.

27.6 Safer Function Wrappers

- `std::move_only_function` replaces `std::function` for **non-copyable callables**.
- `std::bind_back` simplifies **partial function application** with fewer copies.

```cpp
#include <functional>

auto fn = std::move_only_function<int(int)>([](int x){ return x*2; });
```

Benefit: Reduces **heap allocations** and enables safer handling of **move-only types**.

27.7 Strategies for Writing Safer and Faster Code

1. **Audit legacy code** for raw pointers, manual memory management, and loops.
2. **Replace with modern containers** (`std::vector`, `mdspan`, `flat_map`) for speed.
3. **Use `std::expected`** for error handling instead of error codes.
4. **Leverage compile-time computation** with `constexpr` where possible.
5. **Apply ranges and algorithms** instead of manual iteration.
6. **Use type-safe I/O** (`std::print`, `spanstream`) to prevent buffer issues.
7. **Adopt modern function wrappers** (`move_only_function`, `bind_back`) to improve safety.
8. **Refactor string handling** to use safer `std::string_view` operations.

27.8 Example: Modernizing a Legacy Function

Legacy C++20 version:

```
int sum_even(const int* arr, size_t size) {
    int sum = 0;
    for(size_t i=0; i<size; ++i) {
        if(arr[i] % 2 == 0) sum += arr[i];
    }
    return sum;
}
```

C++23 Modern Version:

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

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

- Safer (avoids raw pointer issues)
- Faster (leverages ranges for filtering and iteration)
- More readable and maintainable

27.9 Summary

C++23 empowers developers to **write safer, faster, and more maintainable code** by:

- Introducing **type-safe error handling** (`std::expected`).

- Enhancing **memory safety** (`out_ptr`, `allocate_at_least`).
- Providing **high-performance containers** (`mdspan`, `flat_map`, `flat_set`).
- Expanding **compile-time computation** with `constexpr`.
- Optimizing **loops and transformations** with **ranges**.
- Enabling **safe, concise, and efficient I/O** (`std::print`, `spanstream`).

By adopting these features, developers can **modernize both new and legacy codebases**, ensuring correctness, efficiency, and maintainability in C++23 projects.

Chapter 28

Best Practices for Professional Projects

Developing professional-grade C++ projects requires not only mastery of language syntax but also disciplined approaches to **code safety, maintainability, performance, and team collaboration**. C++23 introduces several features and improvements that can significantly enhance professional workflows. This chapter outlines **best practices for leveraging C++23 in professional projects**, covering design principles, code modernization, library usage, testing, and deployment strategies.

28.1 Embrace Modern Language Features

Professional projects should adopt **modern C++23 features** to improve readability, safety, and performance:

- `std::expected` for type-safe error handling instead of raw error codes.

- **`std::mdspan`** and **flat containers** for efficient and cache-friendly data management.
- **`move_only_function`** and **`bind_back`** for safer function abstractions.
- **Expanded `constexpr`** support for compile-time computations.

Best Practice: Gradually refactor legacy constructs to modern equivalents while maintaining backward compatibility.

28.2 Efficient and Safe Memory Management

- Use **smart pointers** (`std::unique_ptr`, `std::shared_ptr`) instead of raw pointers.
- Employ **`std::out_ptr`** and **`std::inout_ptr`** when interfacing with legacy APIs.
- Use **`allocate_at_least`** and **`start_lifetime_as`** for precise memory control when needed.

Best Practice: Avoid manual memory management; prefer RAII patterns and modern C++23 utilities to reduce leaks and undefined behavior.

28.3 Leverage Ranges and Algorithms

C++23 introduces powerful **range adaptors, constrained algorithms, and conversion utilities**:

- **Range adaptors**: `zip`, `chunk_by`, `adjacent`, `filter`, `transform`.
- **Constrained algorithms**: `find_last`, `starts_with`, `ends_with`.

- **Container conversion**: `ranges::to`.

Best Practice: Replace manual loops and temporary containers with **ranges and views** to improve readability and reduce errors.

28.4 Safe and Efficient I/O

C++23 improves string handling and formatted output:

- **`std::print`** and **`std::println`** provide concise, type-safe formatted output.
- **`spanstream`** allows memory-efficient streaming without copying.
- Use **`string_view`** for lightweight and non-owning string access.

Best Practice: Avoid `sprintf` or manual buffer manipulation; adopt modern C++23 I/O and formatting to prevent buffer overflows and undefined behavior.

28.5 Modernizing Legacy Code

Professional projects often have legacy codebases. C++23 enables **incremental modernization**:

1. Replace raw pointers with **smart pointers** and `out_ptr` utilities.
2. Migrate C-style arrays to **`std::vector`, `mdspan`, or flat containers**.
3. Adopt **ranges** for loops and transformations.
4. Use **`std::expected`** for clear error handling.
5. Apply **`constexpr`** wherever possible for compile-time computation.

Best Practice: Introduce modern features gradually; ensure comprehensive **unit testing** to validate functionality after refactoring.

28.6 Compile-Time Safety and Performance

C++23 allows more operations at compile-time:

- `constexpr` constructors and functions for **containers and algorithms**.
- Compile-time evaluation of **bitset, unique_ptr, and mathematical operations**.

Best Practice: Shift computations to compile-time to improve runtime efficiency, reduce errors, and enhance optimization.

28.7 Project Organization and Modularity

- Adopt **modules (`<std>` and `<std.compat>`)** to reduce compile-time dependencies.
- Use **namespaces and structured headers** for maintainable code organization.
- Minimize global state; prefer **RAII and scoped resource management**.

Best Practice: Structure projects for **clear dependency boundaries and scalable maintenance**, leveraging C++23 modules for faster builds.

28.8 Testing, Debugging, and Diagnostics

C++23 provides tools to improve project reliability:

- `<stacktrace>` for capturing runtime call stacks.
- Use `std::span` and `mdspan` to safely pass data to functions without copies.
- Adopt unit testing frameworks and continuous integration pipelines.

Best Practice: Integrate **modern diagnostics and logging** to trace and debug production issues efficiently.

28.9 Performance Considerations

- Prefer **contiguous and cache-friendly containers** (`flat_map`, `flat_set`).
- Use **ranges and algorithms** to reduce temporary copies and loops.
- Apply **`constexpr` evaluation** to move heavy computations to compile-time.
- Minimize heap allocations in hot paths using **`move_only_function`** and span-based approaches.

Best Practice: Profile code and selectively apply **modern C++23 constructs** to achieve **both safety and speed**.

28.10 Documentation and Code Style

- Adopt **consistent code style** across the team.
- Document use of modern C++23 features, including **ranges, memory utilities, and modules**.
- Encourage code reviews focused on **safety, efficiency, and maintainability**.

Best Practice: Treat C++23 features as tools to **clarify intent and enforce correctness**, not just as syntax enhancements.

28.11 Summary

Professional C++23 projects benefit from:

1. **Type-safe error handling** (`std::expected`).
2. **Safe memory management** (`unique_ptr`, `out_ptr`, `allocate_at_least`).
3. **Modern, cache-friendly containers** (`mdspan`, `flat_map`, `flat_set`).
4. **Efficient loops and algorithms** (ranges and constrained algorithms).
5. **Safer and concise I/O** (`std::print`, `spanstream`).
6. **Compile-time computations** with `constexpr`.
7. **Modular and maintainable project structure** using modules and namespaces.
8. **Robust diagnostics and logging** for debugging and monitoring.

By adhering to these best practices, developers can **maximize the benefits of C++23**, writing **safer, faster, and maintainable professional-grade code** that scales effectively for complex projects.

Chapter 29

C++23 in Embedded, Systems, and High-Performance Applications

C++23 is particularly valuable in domains where **performance, safety, and resource control** are critical, such as **embedded systems, operating systems, real-time applications, and high-performance computing (HPC)**. This chapter explores how C++23's features can improve **efficiency, reliability, and maintainability** in these specialized environments.

29.1 Embedded Systems

Embedded applications often run on **resource-constrained devices**, requiring **deterministic behavior, low memory usage, and high performance**. C++23 provides several tools to address these requirements:

1. **Memory-Efficient Containers**

- **`std::flat_map`** and **`std::flat_set`** provide **contiguous memory storage**, improving **cache locality** and reducing memory overhead.
- **`std::mdspan`** offers **multi-dimensional views** without copying data, ideal for signal processing or sensor arrays.

2. **Compile-Time Computation**

- Expanded **`constexpr`** support allows precomputing values at compile time, reducing runtime load:

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}
constexpr auto fact5 = factorial(5); // computed at compile-time
```

3. **Safer Memory Interfacing**

- Use **`std::out_ptr`** and **`std::inout_ptr`** to safely integrate with low-level APIs.
- **`allocate_at_least`** and **`start_lifetime_as`** enable precise memory allocation control in embedded memory pools.

4. **Lightweight Error Handling**

- **`std::expected`** reduces reliance on exceptions, which may be expensive or unavailable in embedded contexts.

29.2 Systems Programming

C++23 improves **kernel-level and OS-level programming** by providing **robust, low-level abstractions**:

1. **Stack Tracing and Diagnostics**
    - `<stacktrace>` allows capturing call stacks for debugging without adding heavy runtime overhead.
2. **Type Safety and Function Wrappers**
    - `std::move_only_function` and `std::bind_back` support safe and efficient function callbacks in systems code.
    - `std::invoke_r` and `std::forward_like` improve generic programming in low-level routines.
3. **Compile-Time Resource Management**
    - `constexpr unique_ptr` and `bitset` computations enable **compile-time resource initialization**, reducing runtime overhead.

29.3 High-Performance Computing (HPC)

High-performance applications, including **numerical simulations, machine learning kernels, and financial computing**, benefit from **C++23 optimizations**:

1. **Contiguous Data Structures**
    - `mdspan` and **flat containers** optimize memory access for **matrix operations and multidimensional data**, reducing cache misses.

2. **Ranges and Algorithms**

   - New **range adaptors** (`zip`, `chunk_by`, `adjacent`) and **constrained algorithms** (`find_last`, `starts_with`) simplify **data transformations** and enable **vectorization-friendly loops**.
   - `ranges::to` converts efficiently between containers with minimal overhead.

3. **Safer and Faster I/O**

   - **`std::print`**, **`std::println`**, and **`spanstream`** provide **type-safe, zero-copy output**, suitable for logging in HPC applications.

4. **Multi-threading and Concurrency**

   - C++23 builds on prior standards' concurrency support (`std::thread`, `std::jthread`, `std::atomic`) and integrates better **memory safety** with smart pointers and `constexpr` initialization.

29.4 Practical Patterns in Embedded and Systems C++23

1. **Replace raw pointers with smart pointers and `out_ptr` utilities** for safe resource management.
2. **Use `mdspan` for multi-dimensional arrays** instead of manual index computations.
3. **Leverage `constexpr` for precomputed tables and constants** to minimize runtime operations.

4. **Adopt flat containers** for read-heavy data structures requiring fast iteration.

5. **Use `std::expected`** for lightweight error handling in environments where exceptions are unsuitable.

6. **Apply ranges and views** for pipeline-style processing of sensor data, logs, or simulation arrays.

7. **Use `std::stacktrace`** for diagnostics in low-level code without large runtime libraries.

29.5 Example: Modernizing Embedded Signal Processing

Legacy C++20 version:

```
int compute_sum(const int* data, size_t size) {
    int sum = 0;
    for(size_t i = 0; i < size; ++i) sum += data[i];
    return sum;
}
```

C++23 version using `mdspan` and ranges:

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

std::vector<int> buffer = {1,2,3,4,5,6};
```

```
std::mdspan<int, std::extents<2,3>> view(buffer.data());
auto sum = std::accumulate(view.begin(), view.end(), 0);
```

Benefits:

- Safe memory access
- Efficient iteration
- Clear multidimensional representation

29.6 Key Takeaways

C++23 empowers embedded, systems, and high-performance applications by:

- **Reducing runtime overhead** via `constexpr`, optimized containers, and ranges.
- **Increasing safety** through type-safe error handling, smart pointers, and modern function wrappers.
- **Improving maintainability** with clearer, modular code using ranges, views, and modules.
- **Enabling diagnostics and performance monitoring** via `stacktrace` and safe logging mechanisms.

Adopting C++23 in these domains allows developers to achieve **highly reliable, fast, and maintainable software**, even in **resource-constrained or mission-critical environments**.

Chapter 30

The Future: Towards C++26 and Beyond

While C++23 introduces significant improvements in safety, performance, and expressiveness, the evolution of C++ continues toward **C++26 and subsequent standards**. Understanding the direction of the language helps developers **future-proof their code, adopt best practices, and anticipate new features**. This chapter explores emerging trends, proposed features, and the likely trajectory of C++ over the next few years.

30.1 Trends Driving Future C++ Development

The C++ committee is focusing on **several key areas** for upcoming standards:

1. **Safety and Memory Management**
    - Increased emphasis on **memory-safe programming** to reduce undefined behavior.

- Potential expansion of **bounds-checked containers, safer pointer abstractions, and optional garbage collection integration**.

2. **Compile-Time Computation**

   - Continued growth of **`constexpr` support** for more algorithms, containers, and I/O operations.
   - Enabling **complex compile-time evaluation** for high-performance, resource-constrained, or embedded environments.

3. **Concurrency and Parallelism**

   - Enhancements to **standard concurrency utilities** and **parallel algorithms**.
   - Improved **task-based parallelism** and easier integration with hardware accelerators, GPUs, and SIMD instructions.

4. **Modularity and Build Performance**

   - Expansion of **module support** for third-party libraries and standard modules to reduce **compile times** in large projects.
   - Encouragement of **header-only module-friendly libraries** to improve dependency management.

5. **Ranges and Algorithms**

   - Further refinement of **ranges, views, and adaptors** to simplify **pipeline-style processing**.
   - Introduction of more **composable and efficient algorithms** that can integrate seamlessly with both sequential and parallel execution.

6. **Networking and Systems Support**
    - Standardization of **networking utilities** (building on `std::net` proposals).
    - Enhanced **low-level system abstractions**, improving C++ applicability in embedded and OS-level programming.

30.2 Proposed and Expected Features in C++26

Although C++26 is still in development, several features are under discussion and may influence future projects:

- **Logging and diagnostics standardization**
    - C++26 may provide a **dedicated logging facility** integrated with the standard library.
- **Extended coroutines**
    - Enhancements for **asynchronous programming** with reduced overhead.
- **Reflection and compile-time introspection**
    - Improved **compile-time reflection** capabilities for libraries and frameworks.
- **Extended memory-safe facilities**
    - New **smart pointer variants**, enhanced container safety, and possible **automatic lifetime management improvements**.
- **Expanded `std::print` and formatting**

  - More expressive, compile-time verified formatting options.

- **Parallel-friendly containers and algorithms**

  - Standardized containers optimized for **SIMD operations and parallel execution**.

30.3 Implications for Developers

1. **Writing Future-Proof Code**

   - **Prefer modular design**: Use modules and namespaces to reduce coupling.
   - **Adopt `constexpr` and compile-time computation** where feasible.
   - **Use type-safe abstractions** (`std::expected`, smart pointers, `move_only_function`) to minimize refactoring for future memory-safe standards.
   - **Leverage ranges and views** for maintainable, pipeline-style transformations.

2. **Preparing for Migration**

   - Begin **modernizing legacy codebases** to use C++23 features; this reduces friction when adopting C++26.
   - Use **standardized containers and algorithms** rather than custom utilities to maintain compatibility.
   - Keep **diagnostics, logging, and testing infrastructure flexible** to accommodate new standard utilities.

30.4 Emerging Philosophies in C++ Development

- **Safety first**: The future of C++ emphasizes **memory and type safety**, making undefined behavior less common.
- **Compile-time power**: Extensive `constexpr` and metaprogramming support enables **highly optimized, predictable code**.
- **Parallel and asynchronous by default**: Concurrency, ranges, and coroutines will become more deeply integrated.
- **Minimal runtime overhead**: Modern C++ prioritizes **zero-cost abstractions** to retain performance advantages.
- **Ecosystem integration**: Standardization of **networking, logging, and system utilities** will reduce reliance on third-party frameworks.

30.5 Preparing Teams and Projects

To fully leverage C++23 and prepare for C++26:

1. **Educate teams** on modern C++23 features and best practices.
2. **Refactor legacy code incrementally**, adopting smart pointers, ranges, and compile-time features.
3. **Invest in modular project architecture** to minimize dependency and compilation issues.
4. **Integrate robust testing and diagnostics** to ensure smooth transition to future standards.

5. **Follow C++ committee proposals** to anticipate upcoming standard changes and plan accordingly.

30.6 Conclusion

C++23 sets the stage for a **safer, faster, and more expressive language**, while C++26 is expected to continue this trajectory by **enhancing safety, compile-time computation, concurrency, and system-level support**.
By adopting C++23 best practices today, developers:

- Improve code **safety and maintainability**.
- Optimize for **performance and memory efficiency**.
- Prepare for **seamless integration with C++26 and beyond**.

C++ remains a **forward-looking, high-performance language** that balances **modern programming needs** with **low-level control**, making it ideal for embedded systems, HPC, and professional-grade applications.

Appendices

Appendix A: Feature Testing Macros (`__cpp_*`)

Feature testing macros are a crucial mechanism in C++ for writing **portable, robust, and future-proof code**. They allow developers to detect whether a compiler supports specific language or library features introduced in a particular C++ standard version. In C++23, these macros continue to provide essential support for conditional compilation, ensuring code compatibility across different compilers and standard revisions.

Purpose of Feature Testing Macros

The C++ committee introduced feature-testing macros to address common challenges in cross-platform and multi-compiler development:

1. **Portability:** Enable code to compile safely on compilers with varying levels of standard support.
2. **Conditional Compilation:** Allow developers to include or exclude code depending on feature availability.
3. **Backward and Forward Compatibility:** Facilitate gradual adoption of new standard features without breaking existing codebases.

4. **Documentation and Maintainability:** Clearly indicate dependencies on specific language or library features.

Example: Using `std::expected` only if the feature is available:

```
#ifdef __cpp_lib_expected
#include <expected>
#endif
```

This ensures the code only attempts to use `std::expected` if the compiler provides support.

Naming and Format

Feature-testing macros follow a **standardized naming convention**:

```
__cpp_<feature_name> : <value>
```

- **`<feature_name>`** – A concise, descriptive name for the feature (e.g., `lib_expected`, `coroutines`).
- **`<value>`** – An integer representing the adoption year and/or revision of the feature, which helps differentiate between multiple versions of a feature.

Example:

```
#ifdef __cpp_constexpr
static_assert(__cpp_constexpr >= 201907, "Requires C++23 constexpr support");
#endif
```

- Here, `201907` indicates the feature version as defined by the ISO standard.

Categories of Feature Testing Macros

Feature-testing macros are divided into two primary categories:

1. **Language Features**

   These macros indicate support for **core language improvements**, including:

Feature	Macro	Description
`constexpr` enhancements	`__cpp_constexpr`	Full support for extended `constexpr` in C++23.
`consteval`	`__cpp_consteval`	Support for immediate functions evaluated at compile-time.
`coroutines`	`__cpp_impl_coroutine`	Support for coroutine primitives and behavior.
`modules`	`__cpp_modules`	Indicates the availability of C++ modules and import syntax.
`template parameter lists`	`__cpp_template_template_args`	Extended template capabilities and parameter handling.

2. **Library Features**

   Macros for **new standard library components** introduced in C++23 include:

Feature	Macro	Description
`std::expected`	`__cpp_lib_expected`	Indicates support for `std::expected` and associated utilities.
`std::mdspan`	`__cpp_lib_mdspan`	Multi-dimensional span container support.
`std::print` and `std::println`	`__cpp_lib_print`	Indicates availability of modern formatted output functions.
`ranges::to` conversion	`__cpp_lib_ranges_to`	Enables `ranges::to` container conversion.
`move_only_function`	`__cpp_lib_move_only_function`	Move-only function wrapper support for safe callbacks.
Stacktrace utilities	`__cpp_lib_stacktrace`	Support for capturing runtime call stacks.
Flat containers	`__cpp_lib_flat_map` / `__cpp_lib_flat_set`	Indicates availability of contiguous associative containers.

Using Feature-Testing Macros in Practice

Feature-testing macros are often used with **preprocessor directives** to adapt code based on compiler support:

```
#if defined(__cpp_lib_print) && (__cpp_lib_print >= 202202L)
std::print("C++23 print is available\n");
#else
std::cout << "Fallback for older standards\n";
#endif
```

Key guidelines for professional usage:

1. **Always check macro values** – They may indicate the version or defect resolution.

2. **Combine with fallback implementations** – Ensure code remains functional even if a feature is unavailable.

3. **Document feature dependencies** – Makes maintenance and upgrades to future standards easier.

Benefits

- **Improves Portability:** Code can safely compile on multiple compilers without modification.

- **Future-Proofs Code:** Ensures compatibility with new revisions of the C++ standard.

- **Encourages Modern Practices:** Developers can gradually adopt C++23 features while maintaining backward compatibility.

- **Supports Conditional Testing in Libraries:** Library authors can provide feature-specific optimizations without breaking client code.

Summary

Feature-testing macros (`__cpp_*`) are an essential tool for **writing modern, portable, and maintainable C++23 code**. They allow developers to:

- Detect language and library features at compile-time.
- Implement conditional compilation for cross-platform compatibility.
- Gradually adopt modern C++ features while ensuring backward compatibility.
- Document feature requirements explicitly, improving code clarity and maintainability.

By integrating feature-testing macros into your projects, you can **leverage the full power of C++23 while maintaining flexibility and robustness**, preparing your codebase for **future standards like C++26 and beyond**.

Appendix B: Compiler Support (MSVC, GCC, Clang, Intel)

One of the key practical considerations for adopting C++23 is understanding **compiler support** for the latest language and library features. Each compiler evolves at its own pace, and support for new standards can vary depending on **release versions, patches, and experimental flags**. This appendix provides a detailed overview of C++23 support across the **major C++ compilers**: Microsoft Visual C++ (MSVC), GNU Compiler Collection (GCC), Clang/LLVM, and Intel C++ Compiler (oneAPI DPC++).

Microsoft Visual C++ (MSVC)

MSVC has been **aggressively adopting C++23 features** in recent versions, particularly from **Visual Studio 2022 version 17.3 and onward**. Key highlights include:

1. **Language Features**
    - **Modules** – Full support with `import` syntax.
    - **Constexpr Enhancements** – Expanded `constexpr` evaluation including containers and `unique_ptr`.
    - **Coroutines** – Fully supported with `co_await`, `co_yield`, and `co_return`.
    - **Deduction guides and template refinements** – Complete support for template template parameters and constrained templates.
2. **Library Features**
    - `std::expected` – Supported via `<expected>` header.

- **`std::mdspan`** – Full support including multi-dimensional indexing and views.
- **`std::print` and `std::println`** – Supported for formatted output.
- **Ranges and algorithms** – All new C++23 range adaptors and `ranges::to` conversion utilities.
- **Stacktrace** – `<stacktrace>` header supported for runtime diagnostics.

3. **Notes**

- MSVC may require the `/std:c++23` compiler flag to enable full C++23 support.
- Some experimental features may require the `/experimental:module` or `/std:c++latest` flags.

GNU Compiler Collection (GCC)

GCC provides **wide C++23 support starting from version 13**. Highlights include:

1. **Language Features**

- **Expanded `constexpr` and `consteval`** – Full support for compile-time evaluation.
- **Coroutines** – Available with `<coroutine>` and standard keywords.
- **Modules** – Experimental support; requires `-fmodules-ts` flag.
- **Concepts and constrained templates** – Fully implemented.

2. **Library Features**

- **`std::expected`** – Fully available.

- **`std::mdspan`** – Implemented with performance optimizations.
- **`std::print` and `std::println`** – Supported from GCC 13.1 onwards.
- **Ranges and adaptors** – Complete set of C++23 range utilities.
- **Stacktrace** – `<stacktrace>` supported on platforms with libbacktrace or equivalent.

3. **Notes**

- GCC often implements library features slightly behind language features; ensure the **libstdc++ version matches the compiler version**.
- Experimental modules support is improving but not yet fully production-ready in all platforms.

Clang / LLVM

Clang supports C++23 features from **Clang 16 onward**, with continued improvements in LLVM 17 and 18.

1. **Language Features**

- **Constexpr and consteval** – Fully supported.
- **Coroutines** – Fully implemented with LLVM coroutine backend.
- **Modules** – Strong support with the `-fmodules` flag.
- **Concepts** – Complete and conforming implementation.

2. **Library Features**

- **`std::expected`** – Available in libc++ with full functionality.

- **`std::mdspan`** – Supported in libc++17 and above.
- **`std::print`** – Implemented in libc++23.
- **Ranges and adaptors** – Comprehensive support for all C++23 additions.
- **Stacktrace** – Supported using platform-specific backends (e.g., libunwind).

3. **Notes**

- Clang's modular and header-unit support makes it **well-suited for large projects**.
- Compiler flags `-std=c++23` or `-std=c++2b` enable the newest features.

Intel C++ Compiler (oneAPI / ICC)

Intel's compiler focuses on **performance-oriented applications** and supports most C++23 features starting with **oneAPI DPC++/C++ Compiler 2023 update**.

1. **Language Features**

- **Constexpr, coroutines, concepts, and modules** – Supported in modern versions.
- **Extended template and type-trait facilities** – Supported with full C++23 compliance.

2. **Library Features**

- **`std::expected`** – Fully implemented.
- **`std::mdspan`** – Optimized for HPC applications, often leveraging vectorization.

- `std::print` – Available for formatted output.
- **Ranges, adaptors, and algorithms** – Implemented for high-performance pipelines.
- **Stacktrace** – Supported, but may require platform-specific configurations.

3. **Notes**

- Intel compilers are often used in **HPC, embedded, and systems programming**, where **vectorization and memory efficiency** are critical.
- It is compatible with MSVC and GCC ABI in many scenarios, facilitating cross-compiler builds.

Practical Guidelines

1. **Check Compiler Version:** Always verify that the compiler version fully supports the desired C++23 feature.
2. **Use Feature-Testing Macros:** Pair compiler checks with `__cpp_*` macros for maximum portability.
3. **Use Standard Flags:** Use `/std:c++23` (MSVC), `-std=c++23` (GCC/Clang), or equivalent to enable the full standard.
4. **Monitor Library Versions:** Some features are implemented in the standard library, not the compiler core; ensure `libstdc++`, `libc++`, or MSVC STL is up-to-date.
5. **Fallbacks for Legacy Compilers:** For older compilers, provide **alternate implementations or disable non-critical features** gracefully.

Summary

Understanding **compiler support** is essential for leveraging C++23 features in professional projects.

- **MSVC** – Strong support, especially in Visual Studio 2022+.
- **GCC** – Full support from version 13; library features may lag slightly.
- **Clang** – Comprehensive support with strong module and header-unit capabilities.
- **Intel (oneAPI)** – Optimized for HPC and embedded systems; supports most C++23 features.

By carefully managing compiler versions and using **feature-testing macros**, developers can **write portable, future-proof C++23 code** that runs reliably across multiple platforms and toolchains.

Appendix C: Cross-compatibility with C23

C++23 continues to evolve alongside **C23**, the latest standard of the C programming language. While C++ and C share a common heritage, differences in features, syntax, and standard libraries mean that **cross-compatibility requires careful consideration**. This appendix provides a comprehensive overview of C++23's compatibility with C23, including strategies for integration, potential pitfalls, and practical techniques for interoperability.

Shared Heritage and Compatibility Goals

C++ has historically maintained **close compatibility with C**, particularly for low-level programming tasks such as:

- Memory management (`malloc`, `free`)
- Basic data types and structures (`struct`, `enum`, arrays)
- Preprocessor macros and directives (`#define`, `#include`)

However, C++ introduces **additional abstractions** such as classes, templates, and exceptions, which are not part of standard C. Cross-compatibility focuses on:

1. Ensuring **C functions and headers** can be used safely in C++ code.
2. Maintaining **binary compatibility** when linking C23 libraries with C++23 code.
3. Avoiding **undefined behavior** due to differences in type conversions or calling conventions.

Interfacing C++23 with C23

1. **Using `extern "C"`**

   To call C23 functions from C++23, wrap the C headers or function declarations with `extern "C"` to disable C++ name mangling:

   ```
   extern "C" {
       #include "example_c23_library.h"
   }
   ```

   This ensures the C++ compiler uses **C-style linkage**, preserving compatibility with the C23 library's symbols.

2. **Data Type Compatibility**

   - **Primitive types** such as `int`, `char`, `float`, `double` are fully compatible between C23 and C++23.
   - **Structs** and **enums** generally maintain binary layout compatibility if no C++-specific features (like virtual functions or inheritance) are used.
   - **Pointers and arrays** are compatible, but care must be taken with `const` and `volatile` qualifiers.

   Example of compatible struct usage:

   ```
   // C23 header
   typedef struct {
       int id;
       double value;
   } DataRecord;
   // C++23 code
   ```

```
extern "C" {
    #include "data_record.h"
}

DataRecord record;
record.id = 1;
record.value = 3.14;
```

3. **Standard Library Interoperability**

   - C23 introduces features such as `stdatomic.h` for atomic operations and enhanced bounds-checked functions (`memcpy_s`, `strcpy_s`).
   - C++23 provides wrappers or equivalents in `<atomic>` and `<cstring>`.
   - Use **C++ abstractions** when possible for safer and more expressive code, but C23 functions can be directly called when necessary.

   Example:

```
#include <atomic>
#include <cstring>
extern "C" {
    #include <string.h> // C23 safe functions
}

std::atomic<int> counter = 0;
char buffer[50];
memcpy_s(buffer, sizeof(buffer), "Hello C23", 9);
```

Potential Pitfalls

1. **Function Overloading** – C does not support overloading; C++ functions with the same name must be carefully exposed using `extern "C"` or renamed.
2. **Default Parameters** – Not supported in C; avoid using them in interfaces shared with C23.
3. **Struct Padding and Alignment** – Ensure both compilers use compatible packing and alignment for shared data structures.
4. **C++ Exceptions** – Exceptions cannot cross C function boundaries; use error codes or `std::expected` instead.
5. **Name Mangling** – Always use `extern "C"` for C23 headers to prevent symbol mismatch.

Best Practices for Cross-Compatibility

- **Separate Interface Layer**: Create a thin wrapper layer between C23 libraries and C++23 code. This isolates differences and simplifies maintenance.
- **Use `constexpr` and Inline Functions**: For C23 constants and macros, prefer `constexpr` in C++23 for better type safety.
- **Leverage Standard Conversion Utilities**: Use `std::bit_cast`, `std::to_underlying`, and other C++23 utilities for safe conversions.
- **Validate Data Layout**: Use `static_assert(sizeof(...))` to ensure struct sizes match between C23 and C++23.
- **Testing Across Compilers**: Compile and test both C23 and C++23 code with different compilers to ensure ABI compatibility.

Practical Use Cases

1. **Embedded Systems** – Many C23 libraries control hardware peripherals; C++23 can safely wrap these with classes and RAII without breaking the ABI.

2. **High-Performance Computing** – Shared memory or atomic operations can be written in C23 and used seamlessly in C++23 pipelines.

3. **Legacy Integration** – Existing C23 codebases can be modernized gradually by wrapping them in C++23 abstractions.

Example wrapper:

```
extern "C" {
    #include "c23_library.h"
}

class SafeWrapper {
public:
    void doAction() {
        c23_function(); // Direct C23 call
    }
};
```

Summary

C++23 and C23 are **complementary standards**:

- **C++23 extends C23 functionality** with object-oriented programming, templates, coroutines, and safer abstractions.

- **Cross-compatibility is achievable** using `extern "C"`, careful data type handling, and standardized calling conventions.
- Following best practices ensures **portable, maintainable, and robust projects**, allowing developers to leverage both standards efficiently.

By understanding the interaction between C23 and C++23, developers can **modernize legacy C code**, integrate high-performance C libraries, and fully utilize modern C++ features without compromising safety or compatibility.

Appendix D: Quick Reference Tables (Language, Library, Removed/Deprecated Features)

C++23 introduces a wide range of **language improvements, new standard library features, and removals/deprecations**. To facilitate **rapid lookup and practical reference**, this appendix provides **concise tables summarizing key features, their categories, and usage notes**. These tables are designed for **developers, educators, and project managers** who need a quick overview without reading detailed chapters.

Language Features

Feature	Description	Macro / Flag	Notes
`constexpr` Enhancements	Extended compile-time evaluation for containers, smart pointers, and more	`__cpp_constexpr`	Enables `constexpr` algorithms and objects at compile time
`consteval`	Immediate functions evaluated at compile time	`__cpp_consteval`	Guarantees compile-time evaluation

Feature	Description	Macro / Flag	Notes
Coroutines	`co_await`, `co_yield`, `co_return` support	`__cpp_impl_coroutine`	Full support for asynchronous and generator-style programming
Modules	Native modular programming with `import` syntax	`__cpp_modules`	Improves compilation speed and encapsulation
Template Constraints / Concepts	Constrained templates and concepts	`__cpp_concepts`	Improves template readability and type safety
`std::move_only_function`	Move-only callable wrapper	`__cpp_lib_move_only_function`	Ideal for safe, non-copyable callbacks

Feature	Description	Macro / Flag	Notes
`std::bind_back`	Partial application of callable objects	`__cpp_lib_bind_back`	Simplifies functional-style programming

Library Features

Feature	Header	Description	Notes
`std::expected`	`<expected>`	Type-safe error handling alternative to exceptions	Returns `value` or `error`
`std::mdspan`	`<mdspan>`	Multi-dimensional span container for array views	Supports slicing and striding
`std::print` / `std::println`	`<print>`	Formatted output functions	Safer and simpler than `printf` or `iostream`
Ranges adaptors (`zip`, `adjacent`, `chunk_by`)	`<ranges>`	Powerful functional-style iteration utilities	Works with standard containers
`ranges::to`	`<ranges>`	Conversion of ranges into containers	Simplifies container transformations

Feature	Header	Description	Notes
Flat containers (`flat_map`, `flat_set`)	`<flat_map>`, `<flat_set>`	Contiguous associative containers	Improved cache locality, faster iteration
Stacktrace	`<stacktrace>`	Capture runtime call stacks for diagnostics	Works with supported platforms
`spanstream`	`<spanstream>`	Stream interface over `std::span`	Efficient memory-based I/O
`std::stdfloat`	`<stdfloat>`	Extended floating-point type support	Includes `float16_t`, `bfloat16_t`, etc.

Removed / Deprecated Features

Feature	Status	Header	Notes
`std::aligned_storage`	Deprecated	`<type_traits>`	Use `std::aligned_alloc` or `alignas` instead
`std::aligned_union`	Deprecated	`<type_traits>`	Replaced by safer alternatives in C++23
`numeric_limits ::has_denorm`	Deprecated	`<limits>`	Use `std::numeric_limits ::has_denorm_lossless` or other new traits
Mixed wide string concatenation	Removed	`<string>`	Ensures safer string operations

Feature	Status	Header	Notes
Non-encodable wide characters	Removed	`<cwchar>`	Avoids undefined behavior with unsupported characters
Garbage collection support	Removed	`<memory>`	No longer part of standard C++

Reverted Deprecations

Feature	Status	Notes
Subscript operator ‘,‘	Reverted	Original behavior restored for consistency
C headers (`<stdio.h>`, `<stdlib.h>`, etc.)	Kept for compatibility	Provides interoperability with legacy C code

Quick Reference Notes

- **Feature-Testing Macros** (`__cpp_*`) remain critical for conditional compilation. Use them to detect support across compilers (MSVC, GCC, Clang, Intel).
- **New headers** (`<expected>`, `<mdspan>`, `<print>`, `<stacktrace>`, `<spanstream>`) simplify adoption of C++23 features.
- **Removed and deprecated features** should be replaced with safer modern alternatives to maintain portability and maintainability.

- **Ranges and algorithms** provide powerful tools for functional-style iteration, reducing boilerplate and improving performance.

Summary

These **quick reference tables** allow developers to:

1. Identify new language and library features in C++23.
2. Understand deprecated or removed features to avoid potential pitfalls.
3. Map features to their corresponding headers, macros, or usage notes.
4. Rapidly determine which features are safe to use across multiple compilers and platforms.

By using these tables, C++23 developers can **write modern, safe, and portable code** efficiently, keeping legacy code integration and forward compatibility in mind.

Appendix E: Further Reading and References

C++23 represents the latest evolution of the C++ language, introducing **new language features, libraries, and best practices**. While this book provides a comprehensive guide, developers seeking deeper understanding, practical examples, or broader historical context can benefit from targeted further reading. This appendix provides curated resources for continued learning, research, and exploration.

Official Standards and Drafts

1. **ISO C++ Standards Committee**
    - The authoritative source for all C++ standards, including the finalized C++23 standard.
    - Provides detailed specifications for language features, library components, and compiler behaviors.
    - Recommended for developers needing precise definitions of syntax, semantics, and library contracts.
2. **Working Drafts and Proposals (WG21 Papers)**
    - C++23 feature proposals and technical papers can be found under the WG21 working group documentation.
    - Useful for understanding **rationale, design decisions, and planned evolution** of C++ features.

Compiler Documentation

Understanding **compiler support and implementation notes** is essential for practical C++23 development:

- **MSVC (Microsoft Visual C++)**
  - Detailed release notes and feature support documentation for Visual Studio 2022 and later.
  - Guides for enabling `/std:c++23` and module support.
- **GCC (GNU Compiler Collection)**
  - GCC C++23 support details, including library implementation notes for `libstdc++`.
  - Guidance for using flags like `-std=c++23` and experimental modules `-fmodules-ts`.
- **Clang / LLVM**
  - LLVM documentation provides coverage of C++23 features, including modules, coroutines, and ranges.
  - Offers performance tips for large-scale builds and cross-platform development.
- **Intel C++ Compiler (oneAPI)**
  - Provides performance-oriented usage examples and guidance for HPC or embedded systems projects.

Books and Technical Guides

1. **Modern C++ Programming Books**
   - Covers practical usage of C++20 and C++23 features, including coroutines, concepts, and ranges.

- Emphasis on safe and efficient coding practices, memory management, and compile-time programming.

2. **C++ Standard Library References**

   - Provides in-depth coverage of all standard library additions in C++23: containers, algorithms, utilities, and I/O facilities.
   - Explains the interaction between new headers such as `<expected>`, `<mdspan>`, `<stacktrace>`, and `<print>`.

3. **Embedded and Systems Programming Guides**

   - Explains practical applications of C++23 in low-level and high-performance systems.
   - Discusses memory alignment, hardware interaction, and cross-platform portability.

Online Resources

- **ISO/IEC WG21 Website**
  - Source of official papers, proposals, and committee discussions regarding the evolution of C++.
  - Useful for developers interested in the **technical rationale** behind new C++23 features.
- **Compiler Vendor Documentation**
  - Online manuals, tutorials, and example projects from MSVC, GCC, Clang, and Intel.

  - Provides practical guidance for implementing C++23 features across multiple platforms.

- **Community and Expert Blogs**

  - Insightful analyses from C++ experts and contributors to the standard.
  - Offers real-world examples, usage patterns, and performance considerations.

Research Papers and Technical Articles

- **Feature Design Papers** – Detailed exploration of design considerations for `std::expected`, ranges, `std::mdspan`, coroutines, and formatting utilities.
- **Benchmark Studies** – Performance analysis of C++23 containers, algorithms, and memory management utilities.
- **Migration Guides** – Studies on migrating from C++20 to C++23 and modernizing legacy code.

Best Practices for Using References

1. **Combine Sources** – Cross-reference standard documents with compiler implementation notes to ensure accurate feature usage.
2. **Practical Examples** – Look for example code demonstrating **real-world usage** of C++23 features.
3. **Stay Updated** – C++ evolves continuously; follow the latest proposals and committee discussions for C++26 and beyond.
4. **Experiment Safely** – Use small-scale test projects to explore new features before adopting them in production.

Summary

Appendix E provides a **curated roadmap for advanced learning** in C++23:

- Start with **official standards and drafts** for authoritative definitions.
- Consult **compiler documentation** for practical implementation and optimization details.
- Read **technical books and guides** for structured learning and in-depth explanations.
- Explore **online resources, blogs, and research papers** for practical insights and expert analyses.

By leveraging these resources, developers can **gain mastery over C++23**, ensure **cross-platform compatibility**, and adopt **best practices for writing modern, efficient, and safe C++ code**.

References

C++23 is the latest evolution of the C++ programming language, integrating numerous **language enhancements, library expansions, and practical improvements**. The information presented in this book is drawn from **authoritative sources, official documentation, and expert analyses**, ensuring accuracy, completeness, and relevance for professional developers, educators, and advanced students.

Official C++ Standards

1. **ISO/IEC 14882:2023 – Programming Language C++**
    - The formal standard for C++23, published by ISO/IEC.
    - Provides precise specifications for language syntax, semantics, library components, and compile-time behavior.
    - Serves as the ultimate reference for language correctness, feature definitions, and compatibility rules.
2. **C++ Standards Committee (WG21) Papers**
    - Technical proposals (papers Pxxxx) documenting new features, rationales, and design discussions.

- Include proposals for **coroutines, modules, `std::expected`, ranges, flat containers, `std::mdspan`, formatting utilities, and more**.
- Offer insight into **why features were added, how they interact with existing standards, and implementation considerations**.

Compiler Documentation

Accurate understanding of feature support, compilation behavior, and performance is reinforced by **vendor documentation**:

- **Microsoft Visual C++ (MSVC)**
  - Visual Studio release notes, C++23 language and library support documentation, `/std:c++23` usage.
- **GNU Compiler Collection (GCC)**
  - GCC 13 and later release notes, feature tracking, and `libstdc++` library implementation guides.
- **Clang / LLVM**
  - Clang 16+ release notes, experimental features, modules support, coroutines, and ranges.
- **Intel oneAPI C++ Compiler**
  - Performance-oriented documentation, high-performance computing (HPC) guidance, and supported C++23 features.

These sources are essential for **practical adoption**, especially when verifying **compiler-specific behavior and performance optimizations**.

Reference Books and Guides

Several authoritative books and technical guides informed the examples, explanations, and best practices presented in this work:

1. **Modern C++ Programming Books**
    - Covering C++20 and C++23 features including coroutines, concepts, ranges, and memory management.
    - Provide practical examples and case studies of modern C++ usage.
2. **Standard Library References**
    - Comprehensive coverage of new containers, algorithms, utility functions, and I/O facilities.
    - Explains headers like `<expected>`, `<mdspan>`, `<print>`, `<stacktrace>`, `<spanstream>`, `<flat_map>`, `<flat_set>`, and `<stdfloat>`.
3. **Embedded and Systems Programming Guides**
    - Practical applications of C++23 in embedded systems, high-performance computing, and low-level programming.

Online Resources

Reliable online sources were used to supplement official standards and printed references:

- **ISO/IEC WG21 Website** – Technical papers, working drafts, and committee discussions.

- **Compiler Vendor Websites** – Implementation notes, tutorials, and examples for MSVC, GCC, Clang, and Intel.
- **Expert Blogs and Articles** – Detailed analyses of C++23 features, real-world usage examples, and migration strategies from C++20.
- **Educational Repositories** – Sample projects and code illustrating coroutines, ranges, `std::mdspan`, and other new utilities.

Historical and Conceptual References

To provide context for **why features were introduced and how C++ evolved**, additional references include:

- Historical C++ standards (C++98, C++03, C++11, C++14, C++17, C++20) for **feature evolution comparisons**.
- Key technical papers by **C++ language designers and committee members** detailing rationale and design decisions.
- Documentation on related C23 features for **cross-compatibility insights**.

Best Practices and Expert Analysis

- Books and articles emphasizing **safe, efficient, and modern C++ programming** were used to derive guidelines and examples throughout this book.
- Studies and benchmarks focusing on **memory management, container performance, ranges, and compile-time programming** influenced the discussion of practical applications and best practices.

Summary

The **information in this book** is based on:

1. **Official standards** – ISO C++23 and WG21 technical papers.
2. **Compiler documentation** – MSVC, GCC, Clang, Intel.
3. **Reference books** – Covering modern C++ programming and library usage.
4. **Online resources and expert articles** – For real-world insights, examples, and adoption guidance.
5. **Historical and cross-language context** – Providing perspective on C++ evolution and C23 compatibility.

By combining these references, the book ensures that **every topic, feature, and example is accurate, authoritative, and relevant** for developers seeking mastery of **C++23**.