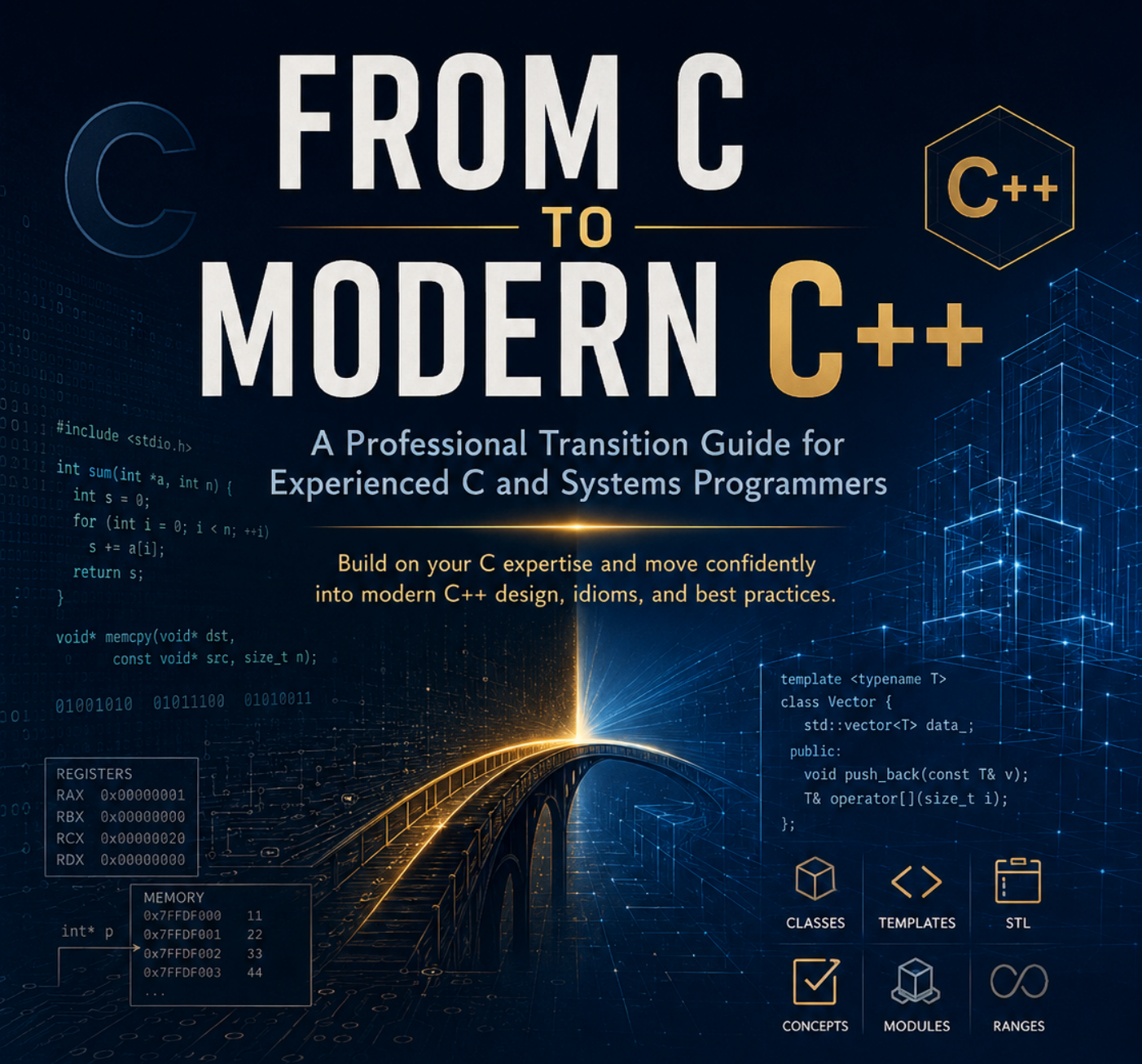
FROM C TO MODERN C++

A Professional Transition Guide for Experienced C and Systems Programmers

Build on your C expertise and move confidently into modern C++ design, idioms, and best practices.



- Build on your C experience
- Master RAII, classes, templates, and the STL
- Understand zero-cost abstractions and design trade-offs
- Use modern C++20 and C++23 features with confidence

Prepared by

Ayman Alheraki

FIRST EDITION · 2026

PRACTICAL • STRUCTURED • LOW-LEVEL AWARE

FRONT MATTER

Copyright and Edition Notice

FRONT MATTER

Technical Scope

The stable core of this book targets C++20 and C++23. C++26 material is explicitly labeled as emerging or experimental because the standard is still under development and compiler support varies. Readers should verify feature-test macros, compiler documentation, and library implementation status before adopting draft features.

FRONT MATTER

Reader Assumptions

The reader is expected to be comfortable with C, pointers, manual resource management, build systems, debugging, binary representation, and systems programming. The book therefore avoids repeating basic programming concepts and concentrates on the conceptual and engineering transition to C++.

FRONT MATTER

Author's Introduction

Why this book was written for experienced C programmers.

I began with C and low-level programming before C++ became my primary language. That path matters because the experienced C programmer does not need another beginner book that explains loops, arrays, or what a pointer is. The real need is a disciplined bridge: what changes, why it changes, what it costs, and how to preserve the clarity and control that systems programmers value.

This book was inspired and strongly encouraged by Fabio Bizzetti, an experienced C and Assembly programmer who described the fragmentation he found when trying to approach modern C++. His request captured a widespread problem: most C++ material either assumes no programming background, dives directly into modern idioms without explaining their foundations, or presents abstractions without connecting them to representation, lifetime, ABI, and machine cost.

The purpose of this work is therefore not to persuade C programmers that C is obsolete. C remains essential. The purpose is to show where C++ can encode invariants, ownership, lifetimes, genericity, and compile-time reasoning more directly - while still allowing the programmer to inspect generated code and control boundaries.

> *"Modern C++ is best learned not by forgetting C, but by **building on its strengths** and correcting its pain points."*

PREPARED BY AYMAN ALHERAKI

FRONT MATTER

Preface

A practical path from C habits to modern C++ mastery.

The safest way to learn C++ is not to translate every C idiom mechanically. Some C patterns remain valid. Some become unnecessary. Others become dangerous when mixed with C++ object lifetime rules. The book repeatedly uses a three-stage method: start with the C mental model, introduce the C++ mechanism, then examine its design rationale and cost model.

Examples favor explicit, small, testable components. Inheritance is not used as the default organizing principle. Smart pointers are not presented as universal replacements for every pointer. Exceptions are discussed alongside explicit result types. Templates are taught as compile-time programming tools, not as syntax to memorize.

The intended result is a conservative but modern C++ style suitable for systems, desktop, embedded, tools, compilers, infrastructure, and performance-sensitive libraries.

FRONT MATTER

How to Use This Book

Read, compare, compile, refactor, and review.

Read Parts I through VI in order. They establish the object, lifetime, ownership, copy, and move model that all later chapters depend on. After that, the parts can be used as focused references.

Compile every laboratory with high warning levels and at least two compilers when portability matters. Inspect optimized assembly for performance-sensitive examples. Run sanitizers during development. Prefer a small set of well-understood language and library features over indiscriminate use of every new facility.

Each lesson contains a C perspective, a C++ transition, a design rationale, a warning or engineering tip, and a compact laboratory. The “Professional Rule” at the end of each lesson summarizes the operational decision to carry into production code.

NAVIGATION

Table of Contents

Click any chapter or appendix title to jump directly to its page. Page numbers are generated from the final paginated edition.

A structured road from established C expertise to confident modern C++ engineering.

18 · Looking toward C++26

PART 01

Reframing the Language

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 1

What C++ Is - and What It Is Not

TRANSITION OBJECTIVE

Understand what c++ is - and what it is not as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

What C++ Is - and What It Is Not should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use what c++ is - and what it is not when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 1: What C++ Is - and What It Is Not

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item1 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to what c++ is - and what it is not. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 2

The C Compatibility Myth

TRANSITION OBJECTIVE

Understand the c compatibility myth as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

The C Compatibility Myth should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use the c compatibility myth when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 2: The C Compatibility Myth

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item2 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to the c compatibility myth. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSC

Zero-Overhead Abstractions

TRANSITION OBJECTIVE

Understand zero-overhead abstractions as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Zero-Overhead Abstractions should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use zero-overhead abstractions when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 3: Zero-Overhead Abstractions

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item3 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to zero-overhead abstractions. Wri down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSC

Value Semantics as the Default

TRANSITION OBJECTIVE

Understand value semantics as the default as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Value Semantics as the Default should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use value semantics as the default when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 4: Value Semantics as the Default

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }
```

```
PacketHeader item4 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to value semantics as the default. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSC

The Cost Model: Source, ABI, and Machine Code

TRANSITION OBJECTIVE

Understand the cost model: source, abi, and machine code as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

The Cost Model: Source, ABI, and Machine Code should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use the cost model: source, abi, and machine code when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 5: The Cost Model: Source, ABI, and Machine Code

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item5 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to the cost model: source, abi, and machine code. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 6

Choosing a C++ Standard and Compiler Mode

TRANSITION OBJECTIVE

Understand choosing a c++ standard and compiler mode as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Choosing a C++ Standard and Compiler Mode should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use choosing a c++ standard and compiler mode when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 6: Choosing a C++ Standard and Compiler Mode

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item6 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to choosing a c++ standard and compiler mode. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 02

A Safer Core Language

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE

DESIGN INTENT

COST MODEL

PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 7

Stronger Types and Better Diagnostics

TRANSITION OBJECTIVE

Understand stronger types and better diagnostics as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Stronger Types and Better Diagnostics should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use stronger types and better diagnostics when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 7: Stronger Types and Better Diagnostics

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item7 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to stronger types and better diagnostics. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 8

References versus Pointers

TRANSITION OBJECTIVE

Understand references versus pointers as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

References versus Pointers should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use references versus pointers when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 8: References versus Pointers

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item8 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to references versus pointers. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 9

const Correctness Revisited

TRANSITION OBJECTIVE

Understand const correctness revisited as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

const Correctness Revisited should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use const correctness revisited when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 9: const Correctness Revisited

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }
```

```
PacketHeader item9 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to const correctness revisited. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 10

auto without Losing Control

TRANSITION OBJECTIVE

Understand auto without losing control as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

auto without Losing Control should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use auto without losing control when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 10: auto without Losing Control

> Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;
```

```
constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item10 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to auto without losing control. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 11

Scoped Enumerations

TRANSITION OBJECTIVE

Understand scoped enumerations as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not

Scoped Enumerations should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use scoped enumerations when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 11: Scoped Enumerations

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>
```

```
struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item11 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to scoped enumerations. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 12

nullptr and Pointer Intent

TRANSITION OBJECTIVE

Understand nullptr and pointer intent as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

nullptr and Pointer Intent should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use nullptr and pointer intent when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 12: nullptr and Pointer Intent

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item12 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to nullptr and pointer intent. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 13

Initialization: Braces, Narrowing, and Order

TRANSITION OBJECTIVE

Understand initialization: braces, narrowing, and order as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Initialization: Braces, Narrowing, and Order should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use initialization: braces, narrowing, and order when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 13: Initialization: Braces, Narrowing, and Order

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item13 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to initialization: braces, narrowing, and order. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 14

Attributes and Compile-Time Intent

TRANSITION OBJECTIVE

Understand attributes and compile-time intent as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Attributes and Compile-Time Intent should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use attributes and compile-time intent when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 14: Attributes and Compile-Time Intent

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item14 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to attributes and compile-time intent. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 03

Functions and Interfaces

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE

DESIGN INTENT

COST MODEL

PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 15

Function Overloading

TRANSITION OBJECTIVE

Understand function overloading as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Function Overloading should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use function overloading when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 15: Function Overloading

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item15 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to function overloading. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 16

Default Arguments and API Stability

TRANSITION OBJECTIVE

Understand default arguments and api stability as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Default Arguments and API Stability should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use default arguments and api stability when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 16: Default Arguments and API Stability

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item16 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to default arguments and api stability. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 17

Inline Functions and the One Definition Rule

TRANSITION OBJECTIVE

Understand inline functions and the one definition rule as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Inline Functions and the One Definition Rule should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use inline functions and the one definition rule when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 17: Inline Functions and the One Definition Rule

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item17 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to inline functions and the one definition rule. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 18

Namespaces and Symbol Organization

TRANSITION OBJECTIVE

Understand namespaces and symbol organization as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Namespaces and Symbol Organization should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use namespaces and symbol organization when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 18: Namespaces and Symbol Organization

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
```

```
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item18 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to namespaces and symbol organization. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 19

Function Objects and Callable Types

TRANSITION OBJECTIVE

Understand function objects and callable types as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and

Function Objects and Callable Types should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use function objects and callable types when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 19: Function Objects and Callable Types

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item19 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to function objects and callable types. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 20

Lambdas from First Principles

TRANSITION OBJECTIVE

Understand lambdas from first principles as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Lambdas from First Principles should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use lambdas from first principles when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 20: Lambdas from First Principles

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
int bias = 4 ;
auto adjust = [bias] ( int value ) noexcept {
 return value + bias ;
 } ;

int result = adjust ( 38 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to lambdas from first principles. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 21

Templates as Compile-Time Functions

TRANSITION OBJECTIVE

Understand templates as compile-time functions as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Templates as Compile-Time Functions should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use templates as compile-time functions when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 21: Templates as Compile-Time Functions

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item21 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to templates as compile-time functions. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 04

Classes without Mysticism

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 22

Classes as Invariants plus Operations

TRANSITION OBJECTIVE

Understand classes as invariants plus operations as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Classes as Invariants plus Operations should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

A class is best understood as a C struct whose valid states and permitted operations are enforced at the definition boundary. It need not imply a hierarchy, framework, or heavy runtime.

WHY C++ DESIGNED IT THIS WAY

The reason for the design is local enforcement of invariants. The cost is usually zero for ordinary value types; dynamic polymorphism has an explicit dispatch and layout cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use classes as invariants plus operations when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 22: Classes as Invariants plus Operations

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <variant>
#include <string>

using Value = std::variant<int, double, std::string> ;
Value value = 42 ;
std::visit ( [] ( const auto& x ) { /* use x */ } , value ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to classes as invariants plus operations. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 23

Constructors and Destructors

TRANSITION OBJECTIVE

Understand constructors and destructors as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Constructors and Destructors should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

A class is best understood as a C struct whose valid states and permitted operations are enforced at the definition boundary. It need not imply a hierarchy, framework, or heavy runtime.

◇ WHY C++ DESIGNED IT THIS WAY

The reason for the design is local enforcement of invariants. The cost is usually zero for ordinary value types; dynamic polymorphism has an explicit dispatch and layout cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use constructors and destructors when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 23: Constructors and Destructors

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item23 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to constructors and destructors. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 24

Access Control and Encapsulation

TRANSITION OBJECTIVE

Understand access control and encapsulation as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Access Control and Encapsulation should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use access control and encapsulation when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 24: Access Control and Encapsulation

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item24 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to access control and encapsulation. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 25

The this Pointer and Object Identity

TRANSITION OBJECTIVE

Understand the this pointer and object identity as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

The this Pointer and Object Identity should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use the this pointer and object identity when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 25: The this Pointer and Object Identity

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item25 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to the this pointer and object identity. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 26

Member Initialization Lists

TRANSITION OBJECTIVE

Understand member initialization lists as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Member Initialization Lists should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use member initialization lists when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 26: Member Initialization Lists

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }
```

```
PacketHeader item26 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to member initialization lists. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 27

Static Data and Static Member Functions

TRANSITION OBJECTIVE

Understand static data and static member functions as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Static Data and Static Member Functions should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use static data and static member functions when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 27: Static Data and Static Member Functions

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
```

```
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item27 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to static data and static member functions. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 28

Operator Overloading with Restraint

TRANSITION OBJECTIVE

Understand operator overloading with restraint as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Operator Overloading with Restraint should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The reason for the design is local enforcement of invariants. The cost is usually zero for ordinary value types; dynamic polymorphism has an explicit dispatch and layout cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use operator overloading with restraint when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 28: Operator Overloading with Restraint

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item28 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to operator overloading with restraint. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 29

Non-Member Interfaces and ADL

TRANSITION OBJECTIVE

Understand non-member interfaces and adl as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Non-Member Interfaces and ADL should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use non-member interfaces and adl when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 29: Non-Member Interfaces and ADL

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item29 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to non-member interfaces and adl. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 05

Resource Management

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 30

RAII: Deterministic Cleanup

TRANSITION OBJECTIVE

Understand raii: deterministic cleanup as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

RAII: Deterministic Cleanup should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use raii: deterministic cleanup when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 30: RAII: Deterministic Cleanup

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
class fd {
 int value_ = -1 ;
public:
 explicit fd ( int v ) noexcept : value_ ( v ) { }
 ~fd ( ) { if ( value_ >= 0 ) ::close ( value_ ) ; }
 fd ( const fd& ) = delete ;
 fd& operator= ( const fd& ) = delete ;
 } ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to raii: deterministic cleanup. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 31

Rule of Zero, Rule of Five

TRANSITION OBJECTIVE

Understand rule of zero, rule of five as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Rule of Zero, Rule of Five should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use rule of zero, rule of five when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 31: Rule of Zero, Rule of Five

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item31 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to rule of zero, rule of five. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 32

unique_ptr for Exclusive Ownership

TRANSITION OBJECTIVE

Understand unique_ptr for exclusive ownership as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

unique_ptr for Exclusive Ownership should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use unique_ptr for exclusive ownership when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 32: unique_ptr for Exclusive Ownership

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <memory>

struct Device {
 void close ( ) noexcept ;
 } ;

auto device = std::make_unique<Device> ( ) ;
device->close ( ) ; // ownership is explicit
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to unique_ptr for exclusive ownership. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 33

shared_ptr and the Cost of Shared Ownership

TRANSITION OBJECTIVE

Understand shared_ptr and the cost of shared ownership as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

shared_ptr and the Cost of Shared Ownership should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use shared_ptr and the cost of shared ownership when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 33: shared_ptr and the Cost of Shared Ownership

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item33 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to shared_ptr and the cost of shared ownership. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 34

weak_ptr and Breaking Cycles

TRANSITION OBJECTIVE

Understand weak_ptr and breaking cycles as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

weak_ptr and Breaking Cycles should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use weak_ptr and breaking cycles when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 34: weak_ptr and Breaking Cycles

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }
```

```
PacketHeader item34 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to weak_ptr and breaking cycles. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 35

Custom Deleters and C Resources

TRANSITION OBJECTIVE

Understand custom deleters and c resources as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Custom Deleters and C Resources should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use custom deleters and c resources when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 35: Custom Deleters and C Resources

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;
```

```
constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item35 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to custom deleters and c resources. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 36

Scope Guards and Transactional Cleanup

TRANSITION OBJECTIVE

Understand scope guards and transactional cleanup as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow,

THE C++ TRANSITION

FROM C TO MODERN C++

Scope Guards and Transactional Cleanup should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use scope guards and transactional cleanup when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 36: Scope Guards and Transactional Cleanup

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item36 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to scope guards and transactional cleanup. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 37

File Descriptors, Handles, and Sockets

TRANSITION OBJECTIVE

Understand file descriptors, handles, and sockets as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

File Descriptors, Handles, and Sockets should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use file descriptors, handles, and sockets when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 37: File Descriptors, Handles, and Sockets

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
class fd {
 int value_ = -1 ;
public:
 explicit fd ( int v ) noexcept : value_ ( v ) { }
 ~fd ( ) { if ( value_ >= 0 ) ::close ( value_ ) ; }
 fd ( const fd& ) = delete ;
 fd& operator= ( const fd& ) = delete ;
 } ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to file descriptors, handles, and sockets. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 06

Copying, Moving, and Lifetime

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 38

Copy Construction and Copy Assignment

TRANSITION OBJECTIVE

Understand copy construction and copy assignment as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Copy Construction and Copy Assignment should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use copy construction and copy assignment when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 38: Copy Construction and Copy Assignment

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item38 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to copy construction and copy assignment. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 39

Move Semantics without Myths

TRANSITION OBJECTIVE

Understand move semantics without myths as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Move Semantics without Myths should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use move semantics without myths when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 39: Move Semantics without Myths

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <utility>
#include <vector>

std::vector<int> build ( ) ;
auto a = build ( ) ; // elision
auto b = std::move ( a ) ; // a remains valid, state unspecified
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to move semantics without myths. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 40

Perfect Forwarding and Reference Collapsing

TRANSITION OBJECTIVE

Understand perfect forwarding and reference collapsing as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Perfect Forwarding and Reference Collapsing should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use perfect forwarding and reference collapsing when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 40: Perfect Forwarding and Reference Collapsing

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item40 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to perfect forwarding and reference collapsing. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 41

Object Lifetime and Storage Duration

TRANSITION OBJECTIVE

Understand object lifetime and storage duration as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Object Lifetime and Storage Duration should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use object lifetime and storage duration when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 41: Object Lifetime and Storage Duration

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item41 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to object lifetime and storage duration. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 42

Temporary Objects and Lifetime Extension

TRANSITION OBJECTIVE

Understand temporary objects and lifetime extension as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Temporary Objects and Lifetime Extension should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use temporary objects and lifetime extension when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 42: Temporary Objects and Lifetime Extension

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
```

```
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item42 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to temporary objects and lifetime extension. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 43

Views, Borrowing, and Dangling

TRANSITION OBJECTIVE

Understand views, borrowing, and dangling as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

i C PERSPECTIVE

Your C knowledge remains the ~~foundation: control flow,~~ representation, ABI, memory, and

THE C++ TRANSITION

Views, Borrowing, and Dangling should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use views, borrowing, and dangling when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 43: Views, Borrowing, and Dangling

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```cpp
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item43 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to views, borrowing, and dangling. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 44

std::move Is a Cast, Not a Move

TRANSITION OBJECTIVE

Understand std::move is a cast, not a move as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

std::move Is a Cast, Not a Move should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use std::move is a cast, not a move when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 44: std::move Is a Cast, Not a Move

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <utility>
#include <vector>

std::vector<int> build ( ) ;
auto a = build ( ) ; // elision
auto b = std::move ( a ) ; // a remains valid, state unspecified
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to std::move is a cast, not a move. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 45

Return Value Optimization and Copy Elision

TRANSITION OBJECTIVE

Understand return value optimization and copy elision as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Return Value Optimization and Copy Elision should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use return value optimization and copy elision when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 45: Return Value Optimization and Copy Elision

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item45 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to return value optimization and copy elision. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 07

The Standard Library Mindset

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE

DESIGN INTENT

COST MODEL

PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 46

std::string versus Character Buffers

TRANSITION OBJECTIVE

Understand std::string versus character buffers as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

std::string versus Character Buffers should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use std::string versus character buffers when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 46: std::string versus Character Buffers

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item46 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to std::string versus character buffers. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 47

std::string_view for Non-Owning Text

TRANSITION OBJECTIVE

Understand std::string_view for non-owning text as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

std::string_view for Non-Owning Text should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use std::string_view for non-owning text when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 47: std::string_view for Non-Owning Text

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <string_view>

void log_line ( std::string_view text ) {
 // no allocation ;  caller retains ownership
 }

log_line ( "ready" ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to std::string_view for non-owning text. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 48

std::vector as the Default Dynamic Array

TRANSITION OBJECTIVE

Understand std::vector as the default dynamic array as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

std::vector as the Default Dynamic Array should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use std::vector as the default dynamic array when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 48: std::vector as the Default Dynamic Array

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <vector>

std::vector<int> samples ;
samples.reserve ( 1024 ) ;
samples.push_back ( 42 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to std::vector as the default dynamic array. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 49

std::array for Fixed-Size Storage

TRANSITION OBJECTIVE

Understand std::array for fixed-size storage as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

std::array for Fixed-Size Storage should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use std::array for fixed-size storage when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 49: std::array for Fixed-Size Storage

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item49 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to std::array for fixed-size storage. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 50

deque, list, and forward_list

TRANSITION OBJECTIVE

Understand deque, list, and forward_list as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

deque, list, and forward_list should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use deque, list, and forward_list when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 50: deque, list, and forward_list

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item50 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to deque, list, and forward_list. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 51

Associative Containers

TRANSITION OBJECTIVE

Understand associative containers as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Associative Containers should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use associative containers when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 51: Associative Containers

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item51 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to associative containers. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 52

unordered Containers and Hashing

TRANSITION OBJECTIVE

Understand unordered containers and hashing as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

unordered Containers and Hashing should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use unordered containers and hashing when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 52: unordered Containers and Hashing

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }
```

```
PacketHeader item52 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to unordered containers and hashing. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 53

Iterators as Generalized Pointers

TRANSITION OBJECTIVE

Understand iterators as generalized pointers as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Iterators as Generalized Pointers should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use iterators as generalized pointers when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 53: Iterators as Generalized Pointers

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;
```

```
constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item53 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to iterators as generalized pointers. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 08

Generic Programming

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 54

Algorithms over Hand-Written Loops

TRANSITION OBJECTIVE

Understand algorithms over hand-written loops as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Algorithms over Hand-Written Loops should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use algorithms over hand-written loops when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 54: Algorithms over Hand-Written Loops

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item54 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to algorithms over hand-written loops. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 55

Function Templates

TRANSITION OBJECTIVE

Understand function templates as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Function Templates should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

◇ WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use function templates when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 55: Function Templates

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item55 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to function templates. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 56

Class Templates

TRANSITION OBJECTIVE

Understand class templates as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Class Templates should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use class templates when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 56: Class Templates

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item56 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to class templates. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 57

Template Argument Deduction

TRANSITION OBJECTIVE

Understand template argument deduction as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Template Argument Deduction should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

◇ WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use template argument deduction when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 57: Template Argument Deduction

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item57 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to template argument deduction. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 58

Specialization and Overload Selection

TRANSITION OBJECTIVE

Understand specialization and overload selection as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Specialization and Overload Selection should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use specialization and overload selection when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 58: Specialization and Overload Selection

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item58 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to specialization and overload selection. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 59

Concepts and Constraints

TRANSITION OBJECTIVE

Understand concepts and constraints as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Concepts and Constraints should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

◇ WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use concepts and constraints when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 59: Concepts and Constraints

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <concepts>

template<class T>
concept Word = std::integral<T> && ( sizeof ( T ) >= 2 ) ;

template<Word T>
constexpr T rotate_left ( T value, unsigned n ) {
 return static_cast<T> ( ( value << n ) | ( value >> ( sizeof ( T ) *8-n ) ) ) ;
 }
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to concepts and constraints. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 60

requires Expressions

TRANSITION OBJECTIVE

Understand requires expressions as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

requires Expressions should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use requires expressions when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 60: requires Expressions

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <concepts>

template<class T>
concept Word = std::integral<T> && ( sizeof ( T ) >= 2 ) ;

template<Word T>
constexpr T rotate_left ( T value, unsigned n ) {
 return static_cast<T> ( ( value << n ) | ( value >> ( sizeof ( T ) *8-n ) ) ) ;
 }
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to requires expressions. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 61

Type Traits and Compile-Time Queries

TRANSITION OBJECTIVE

Understand type traits and compile-time queries as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Type Traits and Compile-Time Queries should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

◇ WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use type traits and compile-time queries when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 61: Type Traits and Compile-Time Queries

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
```

```
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item61 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to type traits and compile-time queries. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 09

Errors and Results

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 62

Policy-Based Design

TRANSITION OBJECTIVE

Understand policy-based design as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Policy-Based Design should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use policy-based design when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 62: Policy-Based Design

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item62 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to policy-based design. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 63

Variadic Templates and Parameter Packs

TRANSITION OBJECTIVE

Understand variadic templates and parameter packs as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Variadic Templates and Parameter Packs should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

◇ WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use variadic templates and parameter packs when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 63: Variadic Templates and Parameter Packs

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item63 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to variadic templates and parameter packs. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 64

Exceptions: Mechanics and Cost

TRANSITION OBJECTIVE

Understand exceptions: mechanics and cost as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Exceptions: Mechanics and Cost should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use exceptions: mechanics and cost when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 64: Exceptions: Mechanics and Cost

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item64 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to exceptions: mechanics and cost. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 65

Exception Safety Guarantees

TRANSITION OBJECTIVE

Understand exception safety guarantees as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Exception Safety Guarantees should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use exception safety guarantees when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 65: Exception Safety Guarantees

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item65 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to exception safety guarantees. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 66

noexcept and Optimization

TRANSITION OBJECTIVE

Understand noexcept and optimization as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

noexcept and Optimization should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use noexcept and optimization when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 66: noexcept and Optimization

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item66 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to noexcept and optimization. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 67

Error Codes Done Well

TRANSITION OBJECTIVE

Understand error codes done well as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Error Codes Done Well should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use error codes done well when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 67: Error Codes Done Well

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item67 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to error codes done well. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 68

std::optional for Maybe-a-Value

TRANSITION OBJECTIVE

Understand std::optional for maybe-a-value as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

std::optional for Maybe-a-Value should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use std::optional for maybe-a-value when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 68: std::optional for Maybe-a-Value

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <optional>

std::optional<int> find_index ( int key ) {
 if ( key == 7 ) return 3 ;
 return std::nullopt ;
 }
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to std::optional for maybe-a-value. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 69

std::variant for Closed Alternatives

TRANSITION OBJECTIVE

Understand std::variant for closed alternatives as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

std::variant for Closed Alternatives should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use std::variant for closed alternatives when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 69: std::variant for Closed Alternatives

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <variant>
#include <string>

using Value = std::variant<int, double, std::string> ;
Value value = 42 ;
std::visit ( [] ( const auto& x ) { /* use x */ } , value ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to std::variant for closed alternatives. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 10

Modern Control and Compile Time

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 70

std::expected for Explicit Failure

TRANSITION OBJECTIVE

Understand std::expected for explicit failure as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

std::expected for Explicit Failure should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use std::expected for explicit failure when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 70: std::expected for Explicit Failure

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <expected>
#include <string>

std::expected<int, std::string> parse_port ( std::string_view s ) {
 if ( s.empty ( ) ) return std::unexpected ( "empty port" ) ;
 return 8080 ;
 }
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to std::expected for explicit failure. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 71

Assertions, Contracts, and Defensive Checks

TRANSITION OBJECTIVE

Understand assertions, contracts, and defensive checks as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Assertions, Contracts, and Defensive Checks should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use assertions, contracts, and defensive checks when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 71: Assertions, Contracts, and Defensive Checks

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item71 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to assertions, contracts, and defensive checks. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 72

Range-Based for

TRANSITION OBJECTIVE

Understand range-based for as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Range-Based for should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use range-based for when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 72: Range-Based for

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

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

std::vector<int> data { 1,2,3,4,5,6 } ;
auto even = data | std::views::filter ( [] ( int x ) { return x % 2 == 0 ; } ) ;
for ( int x : even ) { /* lazy traversal */ }
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to range-based for. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 73

Structured Bindings

TRANSITION OBJECTIVE

Understand structured bindings as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Structured Bindings should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use structured bindings when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 73: Structured Bindings

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item73 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to structured bindings. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 74

if constexpr

TRANSITION OBJECTIVE

Understand if constexpr as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

if constexpr should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

◇ WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use if constexpr when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 74: if constexpr

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
consteval unsigned mask ( unsigned bit ) {
 return 1u << bit ;
 }

constexpr unsigned ready = mask ( 5 ) ;
static_assert ( ready == 0x20 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to if constexpr. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 75

constexpr from Functions to Systems

TRANSITION OBJECTIVE

Understand constexpr from functions to systems as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

constexpr from Functions to Systems should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

◇ WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use constexpr from functions to systems when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 75: constexpr from Functions to Systems

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
consteval unsigned mask ( unsigned bit ) {
 return 1u << bit ;
 }

constexpr unsigned ready = mask ( 5 ) ;
static_assert ( ready == 0x20 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to constexpr from functions to systems. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 76

consteval and Immediate Functions

TRANSITION OBJECTIVE

Understand consteval and immediate functions as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

consteval and Immediate Functions should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

◇ WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use consteval and immediate functions when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 76: consteval and Immediate Functions

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
consteval unsigned mask ( unsigned bit ) {
 return 1u << bit ;
 }

constexpr unsigned ready = mask ( 5 ) ;
static_assert ( ready == 0x20 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to consteval and immediate functions. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 77

Compile-Time Tables and Parsers

TRANSITION OBJECTIVE

Understand compile-time tables and parsers as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Compile-Time Tables and Parsers should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use compile-time tables and parsers when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 77: Compile-Time Tables and Parsers

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item77 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to compile-time tables and parsers. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 11

Memory and Low-Level Control

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 78

Fold Expressions

TRANSITION OBJECTIVE

Understand fold expressions as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Fold Expressions should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use fold expressions when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 78: Fold Expressions

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item78 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to fold expressions. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 79

Pattern-Like Visitation with variant

TRANSITION OBJECTIVE

Understand pattern-like visitation with variant as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Pattern-Like Visitation with variant should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use pattern-like visitation with variant when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 79: Pattern-Like Visitation with variant

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <variant>
#include <string>

using Value = std::variant<int, double, std::string> ;
Value value = 42 ;
std::visit ( [] ( const auto& x ) { /* use x */ } , value ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to pattern-like visitation with variant. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 80

Object Representation and Padding

TRANSITION OBJECTIVE

Understand object representation and padding as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Object Representation and Padding should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use object representation and padding when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 80: Object Representation and Padding

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item80 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to object representation and padding. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 81

Alignment and alignas

TRANSITION OBJECTIVE

Understand alignment and alignas as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Alignment and alignas should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use alignment and alignas when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 81: Alignment and alignas

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item81 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to alignment and alignas. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 82

Placement new and Explicit Lifetime

TRANSITION OBJECTIVE

Understand placement new and explicit lifetime as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Placement new and Explicit Lifetime should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use placement new and explicit lifetime when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 82: Placement new and Explicit Lifetime

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item82 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to placement new and explicit lifetime. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 83

std::byte and Raw Storage

TRANSITION OBJECTIVE

Understand std::byte and raw storage as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

std::byte and Raw Storage should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use std::byte and raw storage when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 83: std::byte and Raw Storage

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item83 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to std::byte and raw storage. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 84

span for Safe Contiguous Views

TRANSITION OBJECTIVE

Understand span for safe contiguous views as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

span for Safe Contiguous Views should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use span for safe contiguous views when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 84: span for Safe Contiguous Views

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <span>

void checksum ( std::span<const std::byte> bytes ) {
 for ( std::byte b : bytes ) { /* ... */ }
 }
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to span for safe contiguous views. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 85

Bit Operations and <bit>

TRANSITION OBJECTIVE

Understand bit operations and <bit> as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Bit Operations and <bit> should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use bit operations and <bit> when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 85: Bit Operations and <bit>

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }
```

```
PacketHeader item85 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to bit operations and <bit>. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 12

Concurrency

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 86

Endianness and Serialization

TRANSITION OBJECTIVE

Understand endianness and serialization as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Endianness and Serialization should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use endianness and serialization when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 86: Endianness and Serialization

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item86 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to endianness and serialization. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 87

Memory-Mapped I/O Boundaries

TRANSITION OBJECTIVE

Understand memory-mapped i/o boundaries as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Memory-Mapped I/O Boundaries should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use memory-mapped i/o boundaries when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 87: Memory-Mapped I/O Boundaries

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item87 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to memory-mapped i/o boundaries. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 88

Threads and jthread

TRANSITION OBJECTIVE

Understand threads and jthread as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Threads and jthread should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C experience with threads, atomics, and platform APIs remains essential. C++ adds portable vocabulary types and a formal memory model that makes synchronization part of the language contract.

◇ WHY C++ DESIGNED IT THIS WAY

The abstraction is useful only when you still reason about contention, cache lines, ordering, cancellation, and shutdown. The library cannot repeal hardware realities.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use threads and jthread when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 88: Threads and jthread

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <atomic>
#include <thread>

std::atomic<unsigned> packets { 0 } ;
std::jthread worker ( [&] ( std::stop_token stop ) {
 while ( !stop.stop_requested ( ) ) { ++packets ; }
 } ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to threads and jthread. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 89

Mutexes and Lock Guards

TRANSITION OBJECTIVE

Understand mutexes and lock guards as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Mutexes and Lock Guards should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C experience with threads, atomics, and platform APIs remains essential. C++ adds portable vocabulary types and a formal memory model that makes synchronization part of the language contract.

◇ WHY C++ DESIGNED IT THIS WAY

The abstraction is useful only when you still reason about contention, cache lines, ordering, cancellation, and shutdown. The library cannot repeal hardware realities.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use mutexes and lock guards when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 89: Mutexes and Lock Guards

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <atomic>
#include <thread>

std::atomic<unsigned> packets { 0 } ;
std::jthread worker ( [&] ( std::stop_token stop ) {
 while ( !stop.stop_requested ( ) ) { ++packets ; }
 } ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to mutexes and lock guards. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 90

Condition Variables

TRANSITION OBJECTIVE

Understand condition variables as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Condition Variables should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use condition variables when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 90: Condition Variables

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item90 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to condition variables. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 91

Atomics and the Memory Model

TRANSITION OBJECTIVE

Understand atomics and the memory model as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Atomics and the Memory Model should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use atomics and the memory model when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 91: Atomics and the Memory Model

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <atomic>
#include <thread>

std::atomic<unsigned> packets { 0 } ;
std::jthread worker ( [&] ( std::stop_token stop ) {
 while ( !stop.stop_requested ( ) ) { ++packets ; }
 } ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to atomics and the memory model. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 92

Lock-Free Is Not Automatically Faster

TRANSITION OBJECTIVE

Understand lock-free is not automatically faster as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Lock-Free Is Not Automatically Faster should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use lock-free is not automatically faster when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 92: Lock-Free Is Not Automatically Faster

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item92 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to lock-free is not automatically faster. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 93

Futures, Promises, and async

TRANSITION OBJECTIVE

Understand futures, promises, and async as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Futures, Promises, and async should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C experience with threads, atomics, and platform APIs remains essential. C++ adds portable vocabulary types and a formal memory model that makes synchronization part of the language contract.

◇ WHY C++ DESIGNED IT THIS WAY

The abstraction is useful only when you still reason about contention, cache lines, ordering, cancellation, and shutdown. The library cannot repeal hardware realities.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use futures, promises, and async when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 93: Futures, Promises, and async

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item93 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to futures, promises, and async. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 13

Modules, Builds, and ABI

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 94

Stop Tokens and Cooperative Cancellation

TRANSITION OBJECTIVE

Understand stop tokens and cooperative cancellation as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Stop Tokens and Cooperative Cancellation should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use stop tokens and cooperative cancellation when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 94: Stop Tokens and Cooperative Cancellation

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item94 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to stop tokens and cooperative cancellation. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 95

Thread-Safe Initialization

TRANSITION OBJECTIVE

Understand thread-safe initialization as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Thread-Safe Initialization should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C experience with threads, atomics, and platform APIs remains essential. C++ adds portable vocabulary types and a formal memory model that makes synchronization part of the language contract.

◇ WHY C++ DESIGNED IT THIS WAY

The abstraction is useful only when you still reason about contention, cache lines, ordering, cancellation, and shutdown. The library cannot repeal hardware realities.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use thread-safe initialization when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 95: Thread-Safe Initialization

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <atomic>
#include <thread>

std::atomic<unsigned> packets { 0 } ;
std::jthread worker ( [&] ( std::stop_token stop ) {
 while ( !stop.stop_requested ( ) ) { ++packets ; }
 } ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to thread-safe initialization. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 96

Headers, Translation Units, and ODR

TRANSITION OBJECTIVE

Understand headers, translation units, and odr as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Headers, Translation Units, and ODR should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use headers, translation units, and odr when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 96: Headers, Translation Units, and ODR

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item96 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to headers, translation units, and odr. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 97

Modules: Goals and Current Reality

TRANSITION OBJECTIVE

Understand modules: goals and current reality as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Modules: Goals and Current Reality should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use modules: goals and current reality when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 97: Modules: Goals and Current Reality

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
export module forge.math ;

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

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to modules: goals and current reality. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 98

Precompiled Headers versus Modules

TRANSITION OBJECTIVE

Understand precompiled headers versus modules as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Precompiled Headers versus Modules should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use precompiled headers versus modules when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 98: Precompiled Headers versus Modules

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
export module forge.math ;

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

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to precompiled headers versus modules. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 99

CMake Targets and Usage Requirements

TRANSITION OBJECTIVE

Understand cmake targets and usage requirements as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

CMake Targets and Usage Requirements should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use cmake targets and usage requirements when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 99: CMake Targets and Usage Requirements

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item99 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to cmake targets and usage requirements. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 100

Compiler Warnings as a Design Tool

TRANSITION OBJECTIVE

Understand compiler warnings as a design tool as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Compiler Warnings as a Design Tool should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use compiler warnings as a design tool when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 100: Compiler Warnings as a Design Tool

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item100 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to compiler warnings as a design tool. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 101

Sanitizers and Dynamic Analysis

TRANSITION OBJECTIVE

Understand sanitizers and dynamic analysis as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Sanitizers and Dynamic Analysis should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use sanitizers and dynamic analysis when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 101: Sanitizers and Dynamic Analysis

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }
```

```
PacketHeader item101 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to sanitizers and dynamic analysis. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 14

Interoperability with C

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 102

Static Analysis and clang-tidy

TRANSITION OBJECTIVE

Understand static analysis and clang-tidy as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Static Analysis and clang-tidy should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use static analysis and clang-tidy when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 102: Static Analysis and clang-tidy

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item102 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to static analysis and clang-tidy. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 103

ABI Boundaries and PImpl

TRANSITION OBJECTIVE

Understand abi boundaries and pimpl as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

ABI Boundaries and PImpl should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use abi boundaries and pimpl when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 103: ABI Boundaries and PImpl

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item103 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to abi boundaries and pimpl. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 104

extern C and Language Linkage

TRANSITION OBJECTIVE

Understand extern c and language linkage as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

extern C and Language Linkage should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use extern c and language linkage when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 104: extern C and Language Linkage

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
extern "C" {
 int legacy_open ( const char* path ) ;
 }

class file_handle {
 int fd_ = -1 ;
public:
 explicit file_handle ( const char* p ) : fd_ ( legacy_open ( p ) ) { }
 } ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to extern c and language linkage. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 105

Wrapping a C Library Safely

TRANSITION OBJECTIVE

Understand wrapping a c library safely as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Wrapping a C Library Safely should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use wrapping a c library safely when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 105: Wrapping a C Library Safely

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item105 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to wrapping a c library safely. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 106

Passing Callbacks across the Boundary

TRANSITION OBJECTIVE

Understand passing callbacks across the boundary as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Passing Callbacks across the Boundary should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use passing callbacks across the boundary when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 106: Passing Callbacks across the Boundary

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
```

```
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item106 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to passing callbacks across the boundary. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 107

Opaque Handles and C-Compatible APIs

TRANSITION OBJECTIVE

Understand opaque handles and c-compatible apis as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Opaque Handles and C-Compatible APIs should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use opaque handles and c-compatible apis when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 107: Opaque Handles and C-Compatible APIs

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
class fd {
 int value_ = -1 ;
public:
 explicit fd ( int v ) noexcept : value_ ( v ) { }
 ~fd ( ) { if ( value_ >= 0 ) ::close ( value_ ) ; }
 fd ( const fd& ) = delete ;
 fd& operator= ( const fd& ) = delete ;
 } ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to opaque handles and c-compatible apis. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 108

Allocators across Module Boundaries

TRANSITION OBJECTIVE

Understand allocators across module boundaries as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Allocators across Module Boundaries should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use allocators across module boundaries when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 108: Allocators across Module Boundaries

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
export module forge.math ;

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

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to allocators across module boundaries. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 109

Struct Layout and Binary Compatibility

TRANSITION OBJECTIVE

Understand struct layout and binary compatibility as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Struct Layout and Binary Compatibility should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use struct layout and binary compatibility when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 109: Struct Layout and Binary Compatibility

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item109 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to struct layout and binary compatibility. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 15

Performance Engineering

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 110

Migrating One Module at a Time

TRANSITION OBJECTIVE

Understand migrating one module at a time as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Migrating One Module at a Time should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use migrating one module at a time when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 110: Migrating One Module at a Time

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
export module forge.math ;

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

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to migrating one module at a time. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 111

Keeping a Stable C ABI

TRANSITION OBJECTIVE

Understand keeping a stable c abi as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Keeping a Stable C ABI should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use keeping a stable c abi when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 111: Keeping a Stable C ABI

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item111 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to keeping a stable c abi. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 112

Measurement before Optimization

TRANSITION OBJECTIVE

Understand measurement before optimization as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Measurement before Optimization should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use measurement before optimization when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 112: Measurement before Optimization

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item112 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to measurement before optimization. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 113

Data-Oriented Design in C++

TRANSITION OBJECTIVE

Understand data-oriented design in c++ as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Data-Oriented Design in C++ should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use data-oriented design in c++ when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 113: Data-Oriented Design in C++

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item113 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to data-oriented design in c++. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 114

Cache Locality and Container Choice

TRANSITION OBJECTIVE

Understand cache locality and container choice as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Cache Locality and Container Choice should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use cache locality and container choice when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 114: Cache Locality and Container Choice

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
```

```
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item114 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to cache locality and container choice. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 115

Inlining, Devirtualization, and LTO

TRANSITION OBJECTIVE

Understand inlining, devirtualization, and lto as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

i C PERSPECTIVE

A class is best understood as a C struct whose valid states and permitted operations are enforced

Inlining, Devirtualization, and LTO should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The reason for the design is local enforcement of invariants. The cost is usually zero for ordinary value types; dynamic polymorphism has an explicit dispatch and layout cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use inlining, devirtualization, and lto when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 115: Inlining, Devirtualization, and LTO

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item115 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to inlining, devirtualization, and lto. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 116

Avoiding Hidden Allocations

TRANSITION OBJECTIVE

Understand avoiding hidden allocations as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Avoiding Hidden Allocations should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use avoiding hidden allocations when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 116: Avoiding Hidden Allocations

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item116 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to avoiding hidden allocations. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 117

Small Buffer Optimization

TRANSITION OBJECTIVE

Understand small buffer optimization as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Small Buffer Optimization should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use small buffer optimization when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 117: Small Buffer Optimization

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item117 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to small buffer optimization. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 16

Design for Systems Programmers

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 118

Expression Templates: Benefits and Risks

TRANSITION OBJECTIVE

Understand expression templates: benefits and risks as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Expression Templates: Benefits and Risks should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

In C, genericity is commonly built with macros, void pointers, code generation, or disciplined naming. C++ moves much of that work into the type system and compile-time evaluation.

WHY C++ DESIGNED IT THIS WAY

The design goal is reusable code without surrendering type checking or runtime performance. The trade-off is more demanding diagnostics and the need to control template boundaries.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use expression templates: benefits and risks when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 118: Expression Templates: Benefits and Risks

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item118 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to expression templates: benefits and risks. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 119

Reading Optimized Assembly

TRANSITION OBJECTIVE

Understand reading optimized assembly as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Reading Optimized Assembly should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use reading optimized assembly when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 119: Reading Optimized Assembly

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item119 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to reading optimized assembly. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 120

Composition before Inheritance

TRANSITION OBJECTIVE

Understand composition before inheritance as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Composition before Inheritance should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

A class is best understood as a C struct whose valid states and permitted operations are enforced at the definition boundary. It need not imply a hierarchy, framework, or heavy runtime.

◇ WHY C++ DESIGNED IT THIS WAY

The reason for the design is local enforcement of invariants. The cost is usually zero for ordinary value

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use composition before inheritance when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 120: Composition before Inheritance

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item120 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to composition before inheritance. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 121

Runtime Polymorphism and Virtual Dispatch

TRANSITION OBJECTIVE

Understand runtime polymorphism and virtual dispatch as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Runtime Polymorphism and Virtual Dispatch should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

A class is best understood as a C struct whose valid states and permitted operations are enforced at the definition boundary. It need not imply a hierarchy, framework, or heavy runtime.

◇ WHY C++ DESIGNED IT THIS WAY

The reason for the design is local enforcement of invariants. The cost is usually zero for ordinary value types; dynamic polymorphism has an explicit dispatch and layout cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use runtime polymorphism and virtual dispatch when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 121: Runtime Polymorphism and Virtual Dispatch

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;
```

```
constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item121 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to runtime polymorphism and virtual dispatch. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 122

Static Polymorphism and CRTP

TRANSITION OBJECTIVE

Understand static polymorphism and crtp as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

i C PERSPECTIVE

A class is best understood as a C struct whose valid states and permitted operations are enforced at the definition boundary. It need

Static Polymorphism and CRTP should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The reason for the design is local enforcement of invariants. The cost is usually zero for ordinary value types; dynamic polymorphism has an explicit dispatch and layout cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use static polymorphism and crtp when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 122: Static Polymorphism and CRTP

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
```

```
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item122 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to static polymorphism and crtp. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 123

Type Erasure

TRANSITION OBJECTIVE

Understand type erasure as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

i C PERSPECTIVE

Your C knowledge remains the ~~foundation: control flow,~~ representation, ABI, memory, and

Type Erasure should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use type erasure when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 123: Type Erasure

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```cpp
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item123 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to type erasure. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 124

Dependency Injection without Frameworks

TRANSITION OBJECTIVE

Understand dependency injection without frameworks as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Dependency Injection without Frameworks should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use dependency injection without frameworks when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 124: Dependency Injection without Frameworks

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item124 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to dependency injection without frameworks. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 125

State Machines with Types

TRANSITION OBJECTIVE

Understand state machines with types as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

State Machines with Types should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use state machines with types when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 125: State Machines with Types

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item125 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to state machines with types. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 17

C++20/C++23 Working Practice

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 126

Strong Types for Units and IDs

TRANSITION OBJECTIVE

Understand strong types for units and ids as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Strong Types for Units and IDs should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use strong types for units and ids when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 126: Strong Types for Units and IDs

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item126 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to strong types for units and ids. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 127

API Design for Ownership and Lifetime

TRANSITION OBJECTIVE

Understand api design for ownership and lifetime as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

API Design for Ownership and Lifetime should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

You already understand addresses, storage duration, allocation, and cleanup in C. The C++ step is to encode those facts in types and scopes so that the compiler can enforce more of the protocol.

◇ WHY C++ DESIGNED IT THIS WAY

The feature exists to make lifetime and ownership visible in interfaces, not to hide the machine. Used well, it removes manual cleanup paths while preserving deterministic behavior.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use api design for ownership and lifetime when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 127: API Design for Ownership and Lifetime

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item127 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to api design for ownership and lifetime. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 128

Ranges and Views

TRANSITION OBJECTIVE

Understand ranges and views as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Ranges and Views should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use ranges and views when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 128: Ranges and Views

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

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

std::vector<int> data { 1,2,3,4,5,6 } ;
auto even = data | std::views::filter ( [] ( int x ) { return x % 2 == 0 ; } ) ;
for ( int x : even ) { /* lazy traversal */ }
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to ranges and views. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 129

Formatting with std::format

TRANSITION OBJECTIVE

Understand formatting with std::format as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Formatting with std::format should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use formatting with std::format when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 129: Formatting with std::format

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item129 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to formatting with std::format. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 130

Chrono and Calendar Types

TRANSITION OBJECTIVE

Understand chrono and calendar types as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Chrono and Calendar Types should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use chrono and calendar types when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 130: Chrono and Calendar Types

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }
```

```
PacketHeader item130 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to chrono and calendar types. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 131

Filesystem

TRANSITION OBJECTIVE

Understand filesystem as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Filesystem should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use filesystem when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 131: Filesystem

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;
```

```
constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item131 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to filesystem. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 132

Coroutines: The Language Machinery

TRANSITION OBJECTIVE

Understand coroutines: the language machinery as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow,

Coroutines: The Language Machinery should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use coroutines: the language machinery when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 132: Coroutines: The Language Machinery

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item132 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to coroutines: the language machinery. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 133

Generator-Style Abstractions

TRANSITION OBJECTIVE

Understand generator-style abstractions as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Generator-Style Abstractions should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use generator-style abstractions when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 133: Generator-Style Abstractions

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item133 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to generator-style abstractions. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

PART 18

Looking toward C++26

Purpose: Build a production-minded bridge from established C practice to the C++ mechanisms in this part, with explicit attention to representation, lifetime, diagnostics, portability, and cost.

C PERSPECTIVE | DESIGN INTENT | COST MODEL | PROFESSIONAL PRACTICE

CHAPTER LESSON

LESSON 134

Modern Numerics and SIMD Direction

TRANSITION OBJECTIVE

Understand modern numerics and simd direction as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Modern Numerics and SIMD Direction should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use modern numerics and simd direction when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 134: Modern Numerics and SIMD Direction

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item134 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to modern numerics and simd direction. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 135

C++23 Library Improvements

TRANSITION OBJECTIVE

Understand c++23 library improvements as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

C++23 Library Improvements should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use c++23 library improvements when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 135: C++23 Library Improvements

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item135 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to c++23 library improvements. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 136

C++26 Status and Experimental Modes

TRANSITION OBJECTIVE

Understand c++26 status and experimental modes as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

C++26 Status and Experimental Modes should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use c++26 status and experimental modes when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 136: C++26 Status and Experimental Modes

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
```

```
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item136 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to c++26 status and experimental modes. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 137

Reflection: Promise and Limits

TRANSITION OBJECTIVE

Understand reflection: promise and limits as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and

Reflection: Promise and Limits should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use reflection: promise and limits when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 137: Reflection: Promise and Limits

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```cpp
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item137 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to reflection: promise and limits. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 138

Contracts Direction

TRANSITION OBJECTIVE

Understand contracts direction as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Contracts Direction should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

✓ PROFESSIONAL RULE

Use contracts direction when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 138: Contracts Direction

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item138 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to contracts direction. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 139

Safer Library Interfaces

TRANSITION OBJECTIVE

Understand safer library interfaces as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Safer Library Interfaces should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use safer library interfaces when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 139: Safer Library Interfaces

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item139 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to safer library interfaces. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 140

Execution and Parallelism Direction

TRANSITION OBJECTIVE

Understand execution and parallelism direction as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Execution and Parallelism Direction should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use execution and parallelism direction when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 140: Execution and Parallelism Direction

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item140 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to execution and parallelism direction. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 141

Feature-Test Macros

TRANSITION OBJECTIVE

Understand feature-test macros as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Feature-Test Macros should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST

State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use feature-test macros when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 141: Feature-Test Macros

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item141 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to feature-test macros. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 142

Portable Adoption Strategy

TRANSITION OBJECTIVE

Understand portable adoption strategy as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

Portable Adoption Strategy should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI impact, and toolchain support. A feature that obscures these properties is being used badly, even when the feature itself is sound.

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use portable adoption strategy when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 142: Portable Adoption Strategy

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item142 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to portable adoption strategy. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

CHAPTER LESSON

LESSON 143

A Conservative Modern C++ Profile

TRANSITION OBJECTIVE

Understand a conservative modern c++ profile as an engineering tool rather than a language ornament, and connect it to familiar C mechanisms.

THE C++ TRANSITION

A Conservative Modern C++ Profile should be introduced only after identifying the contract that the C version leaves implicit. In professional C++, the important change is usually not syntax. It is the ability to attach a rule to a type, a scope, a function signature, or a compile-time constraint so that misuse is rejected earlier.

i C PERSPECTIVE

Your C knowledge remains the foundation: control flow, representation, ABI, memory, and operating-system interfaces do not disappear. C++ adds stronger ways to state intent and compose reusable components.

◇ WHY C++ DESIGNED IT THIS WAY

The central question is not whether the syntax is shorter. It is whether the type system can prevent an invalid state, make ownership clear, or move a check from runtime to compile time without unacceptable cost.

! WARNING

Do not adopt a C++ facility merely because it is modern. Verify ownership, lifetime, exception behavior, allocation, code size, ABI

OPERATIONAL CHECKLIST		
State the invariant or protocol this feature is intended to enforce.	Identify ownership, borrowing, copying, moving, and failure behavior.	Check diagnostics, optimized code, and ABI impact on the actual toolchain.

PROFESSIONAL RULE

Use a conservative modern c++ profile when it makes a real contract visible or removes an entire category of cleanup and misuse. Otherwise prefer the smallest mechanism that communicates the design.

APPLIED LABORATORY

Laboratory 143: A Conservative Modern C++ Profile

Compile first with C++23 mode and strong warnings. For experimental C++26 features, use the compiler-specific experimental mode and guard the code with feature-test macros.

FOCUSED EXAMPLE

```
#include <cstdint>

struct PacketHeader {
 std::uint16_t type { } ;
 std::uint16_t size { } ;
 } ;

constexpr bool valid ( PacketHeader h ) noexcept {
 return h.size >= sizeof ( PacketHeader ) ;
 }

PacketHeader item143 { .type = 1, .size = 64 } ;
static_assert ( sizeof ( PacketHeader ) == 4 ) ;
```

WHAT TO INSPECT

Question	Engineering observation
Type safety	Which invalid calls are rejected before linking or execution?
Lifetime	Who owns each object, and how is destruction guaranteed?
Runtime cost	Does the optimized output add branches, allocation, indirection, or synchronization?
Failure model	Can the operation throw, return an error, terminate, or leave a partial state?
ABI	Can this type or function safely cross a C boundary or shared-library boundary?

PRACTICE TASK

Take a small C component from your own work that relates to a conservative modern c++ profile. Write down its invariants, ownership rules, error paths, and ABI constraints. Produce a C++ version that changes only one design dimension at a time, then compare warnings, testability, assembly, and failure behavior.

? REVIEW QUESTION

Could another experienced C programmer understand the ownership and cost model from the interface alone? If not, simplify the design or make the contract more explicit.

REFERENCE SECTION

Conclusion - Becoming a C++ Systems Programmer

The transition from C to C++ is not a rejection of low-level knowledge. It is an attempt to preserve that knowledge in stronger interfaces. The best C++ programmers still understand memory, calling conventions, cache behavior, object representation, system calls, and generated instructions. What changes is how much of the program protocol can be checked by the language and library.

A professional migration should be incremental. Begin with stronger initialization, scoped enums, references where null is invalid, RAII wrappers for resources, standard containers, and clear value semantics. Add templates and concepts where they remove duplication without spreading complexity. Introduce exceptions, coroutines, modules, or advanced metaprogramming only when the surrounding engineering model is ready for them.

The goal is not “maximum C++.” The goal is a carefully chosen C++ subset that improves correctness, maintainability, and expressiveness while preserving predictable performance and explicit system boundaries.

Experienced C programmers already possess the hardest foundation: respect for representation, lifetime, cost, and the machine. Modern C++ becomes productive when it is learned as a way to encode that discipline rather than escape from it.

```
FINAL PRINCIPLE
Use abstraction to expose the real design, not to conceal it. Measure cost, make ownership
explicit, keep boundaries stable, and let the compiler enforce every rule that can be stated
precisely.
```

REFERENCE SECTION

Appendix A - Recommended Compiler Profiles

• GCC/Clang development: -std=c++23 -Wall -Wextra -Wpedantic -Wconversion -Wsign-conversion -Wshadow -Wnon-virtual-dtor -Wold-style-cast.

• Debug and dynamic analysis: -g3 -O1 -fsanitize=address,undefined -fno-omit-frame-pointer.

• Release: choose optimization and LTO only after representative benchmarks; retain symbols or split debug information for postmortem analysis.

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Appendix B - C-to-C++ Migration Checklist

• Compile C sources as C first and remove undefined behavior before translation.

• Wrap resources with deterministic destructors before redesigning algorithms.

• Replace owning raw pointers before replacing all raw pointers.

• Introduce standard containers and spans at module boundaries.

• Keep a stable extern "C" facade when external consumers require it.

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Appendix C - Ownership Vocabulary

- Owner: responsible for ending a resource lifetime.
- Borrower: temporarily observes or uses a resource owned elsewhere.
- View: a lightweight non-owning range or text reference.
- Value: self-contained state copied or moved according to declared semantics.
- Handle: compact identifier whose operations are mediated by another subsystem.

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Appendix D - Exception-Safety Matrix

- No-throw guarantee: operation does not emit an exception.
- Strong guarantee: failure leaves the observable state unchanged.
- Basic guarantee: invariants remain valid and resources do not leak.
- No guarantee: failure may leave the object unusable; avoid in public components.

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Appendix E - Feature Adoption Policy

- Prefer published-standard facilities with mature implementation support.
- Use feature-test macros instead of compiler-version guesses.
- Isolate draft-standard features behind narrow adapters.
- Maintain a fallback path for every toolchain that matters to the product.

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Appendix F - Code Review Questions

- Is ownership visible?
- Can a view outlive its source?
- Are failure paths explicit?
- Does the interface allocate?
- Is dynamic dispatch intentional?
- Is the ABI boundary stable?
- Are units and ranges represented by types?
- Can invalid states be constructed?

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Appendix G - Glossary

- ABI - Application Binary Interface.

- ADL - Argument-dependent lookup.
- ODR - One Definition Rule.
- RAII - Resource Acquisition Is Initialization.
- RVO - Return Value Optimization.
- SBO - Small Buffer Optimization.
- TU - Translation unit.
- UB - Undefined behavior.

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Appendix H - References and Further Study

- ISO/IEC 14882:2024, Programming Languages - C++ (C++23).
- WG21 public papers and current working draft, open-std.org/JTC1/SC22/WG21.
- GCC C++ Standards Support and libstdc++ implementation status.
- Clang C++ Programming Language Status and compiler documentation.
- C++ Core Guidelines, maintained by the C++ community.
- Compiler Explorer for comparative generated-code inspection.
- Platform ABI documentation: System V AMD64 ABI, Microsoft x64 ABI, and AAPCS64.

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Appendix I - A Conservative Production Subset

• Use: value types, RAII, rule of zero, unique ownership, standard containers, spans, string views, algorithms, concepts at API boundaries, expected/optional for explicit results, and disciplined constexpr.

• Use with policy: exceptions, shared ownership, virtual inheritance, coroutines, modules, custom allocators, and advanced metaprogramming.

• Avoid by default: owning raw pointers, naked new/delete, unchecked narrowing, macro-based generic programming, implicit ownership transfer, and undefined-behavior optimizations.

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Appendix J - Thirty-Day Transition Plan

• Days 1-5: initialization, references, const, enums, overloads, namespaces.

• Days 6-10: classes, constructors, destructors, RAII, rule of zero.

• Days 11-15: string, vector, array, span, algorithms, iterators.

• Days 16-20: move semantics, smart pointers, optional, variant, expected.

• Days 21-25: templates, concepts, constexpr, ranges.

• Days 26-30: concurrency, tooling, modules, C interoperability, migration pilot.

```
APPLICATION NOTE
Treat this appendix as a working checklist. Adapt it to the product risk, supported compilers,
target ABIs, and team experience.
```

REFERENCE SECTION

Acknowledgments

Special thanks to Fabio Bizzetti for articulating the need for a C++ book written directly for experienced C and Assembly programmers, and for encouraging the completion of an idea that had already begun to take shape.

Thanks also to the C and C++ communities, compiler engineers, standards contributors, library authors, and systems programmers whose work makes precise, portable, high-performance software possible.

FROM C TO MODERN C++

A professional bridge from machine-aware C programming to disciplined, production-grade modern C++.

Ayman Alheraki | ForgeVM.org | 2026