https://simplifycpp.org

Modern C++ Encyclopedia

PROGRAM STRUCTURE & MODULES

Volume 2

Prepared by Ayman Alheraki

DRAFT

Modern C++ Encyclopedia

PROGRAM STRUCTURE & MODULES

Volume 2

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

Prepared by Ayman Alheraki

Contents

Author's Introduction

This volume of the **Modern C++ Encyclopedia Series** is dedicated to one of the most transformative structural shifts in the history of C++: the transition from header-based architecture to module-based design, spanning C++20 through C++26. The goal of this book is not merely to explain modules syntactically, but to explore what they truly represent — a redefinition of how large-scale C++ systems are organized, built, maintained, and evolved.

For decades, C++ programs have been structured around translation units, header inclusion, and preprocessor-driven composition. While this model enabled extraordinary flexibility and performance, it also introduced structural fragility: hidden dependencies, cyclic includes, compile-time explosions, macro leakage, and One Definition Rule pitfalls. As systems scaled, these issues became architectural rather than stylistic problems. The language evolved; the engineering discipline around it had to evolve as well.

C++20 introduced Modules as a standardized mechanism to address these structural limitations. However, adopting modules is not a cosmetic refactor. It requires a shift in mindset: from file-based organization to boundary-driven design; from textual inclusion to explicit interface contracts; from accidental coupling to deliberate export surfaces. Modules make architecture visible to the compiler. They force clarity. They reward discipline.

This volume was written from the perspective of a systems engineer who has lived through large C++ codebases — legacy migrations, build complexity, ABI constraints, cross-platform demands, and long-lived production systems. It acknowledges that real-world adoption is incremental. Most projects will live for years in hybrid states: headers coexisting with modules,

legacy libraries interacting with new module boundaries, build systems gradually gaining reliable dependency scanning. This book does not ignore that reality; it embraces it and provides a structured path forward.

The central idea of this volume is simple: **modules are an architectural tool**. They are not primarily about faster compilation or modern syntax. They are about expressing intent. A module interface defines what a component promises. An implementation unit defines what it hides. Partitions allow scale without boundary erosion. Imports define explicit relationships. When used correctly, modules turn implicit design into enforceable structure.

C++20 laid the foundation. C++23 refined the ecosystem integration. C++26 continues the maturation of the language and its surrounding tooling. Yet language features alone do not guarantee architectural quality. Engineers must design module boundaries thoughtfully: aligning them with domain concepts, ownership models, and long-term maintainability goals. Poorly designed modules can replicate the same chaos that headers once created. Well-designed modules can transform the clarity of an entire codebase.

This volume therefore focuses on principles before mechanics. It examines how to discover architectural boundaries in existing systems. It explains how to design exported interfaces that remain stable over time. It explores how to eliminate cyclic dependencies by rethinking component relationships. It addresses how build systems must reflect module graphs accurately. And it demonstrates, through a full project reconstruction, how a real application can be restructured using modules without sacrificing production stability.

The intended reader is the serious C++ engineer: someone responsible for systems that matter, for code that lives beyond prototypes, and for teams that depend on clear structure. Whether you are modernizing a legacy system or designing a new architecture from scratch, this volume aims to provide both conceptual clarity and practical guidance.

Modern C++ is no longer only about mastering templates, memory models, or low-level optimization. It is about mastering structure. It is about understanding that architecture is part of correctness. Modules are the language's commitment to that philosophy. This book is my

contribution to helping you use them with precision, discipline, and long-term vision.

Ayman alheraki

Preface

This volume, **PROGRAM STRUCTURE & MODULES (C++20 → C++26)**, is written with one purpose: to help serious C++ engineers design, refactor, and ship large systems with **clean boundaries** and **tool-verifiable interfaces**. It is not a collection of isolated features. It is an engineering guide for moving from a codebase that merely *compiles* to a codebase that is *structurally governed*.

C++ has always been a language where architecture matters. For decades, however, architecture was enforced mostly by convention: directory layout, naming rules, header discipline, include-order folklore, and best-effort review culture. The compilation model was largely textual: headers were copied into translation units; macros could invisibly change meaning; and dependencies were frequently implicit. This model enabled immense flexibility, but at scale it also created predictable failure patterns:

- build graphs that exist in people's minds rather than in tooling,
- transitive dependency explosions driven by header inclusion,
- accidental coupling where internal details become public by inclusion,
- cyclic dependency webs that are difficult to break without rewrites,
- inconsistent configuration that triggers One Definition Rule hazards,

- architectural drift where boundaries erode gradually until the system becomes one large component.

C++20 Modules are the first standardized mechanism in the language that can make architectural intent **explicit and enforceable**. Modules let you declare a component boundary, export an interface, and import only what you depend on. They also force the ecosystem to treat structure as a first-class build concept: dependency scanning, compilation ordering, artifact discovery, and interface distribution become part of correctness. This is why modules must be studied as both **language** and **systems tooling**.

At the same time, this volume is realistic. Most organizations will not flip a switch and become "fully modular." Many codebases will exist for years in hybrid states: some parts remain headers, some become header units, and some become named modules. You will integrate third-party libraries that may not be modular. You will face cross-platform constraints. You will adapt to differences among compiler frontends and build systems. This book is written to help you succeed under those constraints.

What this volume is about

This volume is about designing program structure in a way that scales across:

- **time** (projects that live for years),
- **people** (teams, ownership boundaries, reviews),
- **build pipelines** (CI, caching, incremental builds, dependency scanning),
- **platforms** (Windows/Linux/macOS, multiple compilers),
- **distribution** (internal libraries, SDKs, plugins, binary deliverables).

Modules are central, but not isolated. You will learn to:

- discover architectural seams in existing code,
- create module boundaries that reflect domain and ownership,
- design exported interfaces that remain stable over time,
- eliminate cyclic dependencies by restructuring imports and responsibilities,
- migrate safely using staged approaches and compatibility layers,
- align the build graph with the import graph to make structure enforceable.

Who this volume is written for

This volume targets experienced C++ engineers who:

- maintain medium-to-large production codebases,
- care about long-term maintainability and correctness,
- are responsible for build reliability and developer productivity,
- design libraries consumed by other teams,
- refactor legacy systems without breaking delivery.

If you are a beginner, you can still follow the examples, but the primary audience is the engineer who understands that “architecture is part of correctness.”

How to read and use this volume

This book is designed to be used in two ways:

- **Sequentially**, to build a complete mental model: architecture → boundaries → modules → build integration → migration → full project rebuild.
- **As a reference**, to solve real problems: cyclic imports, exported interface design, partition organization, mixed header/module strategies, and build graph diagnostics.

Each chapter focuses on decisions that matter in real projects:

- what to export and what to hide,
- how to prevent accidental coupling,
- how to keep ownership boundaries enforceable,
- how to keep builds deterministic and scalable.

Key terminology used consistently

To avoid confusion, this volume uses the following terms in a strict engineering sense:

- **Named module**: a module declared with `module M;` or `export module M;`.
- **Module interface unit**: a module unit that exports declarations (typically begins with `export module`).
- **Module implementation unit**: a module unit that provides definitions without exporting them (typically begins with `module`).
- **Partition**: a named slice of a module (e.g., `export module M:part;`), used to scale and organize.
- **Header unit**: an importable representation of a header, used for staged migration and compatibility.

- **Compiled interface artifact**: the build output representing a compiled module interface (toolchain terminology varies; concept is universal).

A practical philosophy for modular C++

This volume follows a pragmatic philosophy:

- **Boundaries first.** A module boundary should represent a real responsibility boundary, not a file grouping.
- **Exports are contracts.** Every exported name increases your API surface; keep it deliberate.
- **Internals must stay internal.** If an internal detail leaks, it becomes hard to remove later.
- **Build correctness is correctness.** If the build graph lies, the program's structure lies.
- **Migration is staged.** We prefer safe incremental paths over heroic rewrites.

You will see these principles applied repeatedly: we will intentionally choose designs that reduce churn, prevent boundary erosion, and keep refactoring costs predictable.

Tooling expectations and reality

C++20 modules are standardized, but implementation and workflow are still toolchain-dependent. Build systems increasingly support module dependency scanning and correct compilation ordering, but your success depends on aligning:

- compiler capabilities,
- build system support,

- IDE and analysis tooling behavior,
- third-party library constraints.

Therefore, examples in this volume are written to be conceptually portable:

- we separate interface, partitions, and implementation cleanly,
- we avoid depending on non-portable hacks,
- we show mixed-mode integration patterns that reflect real projects.

A minimal, correct module pattern

This is the smallest example that still teaches correct structure: an explicit interface, an internal helper, and a consumer that imports only what it needs.

Interface unit

```
// math_api.cppm
export module math.api;

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

Implementation unit

```
// math_impl.cpp
module math.api;

namespace {
    int normalize(int x) { return x; } // internal detail, not exported
}
```

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

Consumer

```
// main.cpp
import math.api;

#include <iostream>

int main() {
    std::cout << add(40, 2) << "\n";
}
```

Even at this scale, the intent is visible:

- the module declares a boundary,
- the interface exports a contract,
- implementation details remain internal,
- the consumer imports, rather than includes, the contract.

Scaling a component with partitions

Partitions let you scale internals and organize a module without turning the primary interface into a monolith. The pattern used throughout this volume is:

- a small primary interface that re-exports carefully,

- partitions that group cohesive responsibilities,
- implementation units that keep details private.

Primary interface re-exporting partitions

```
// net.cppm
export module net;

export import :tcp;
export import :udp;
```

Partition interfaces

```
// net_tcp.cppm
export module net:tcp;
export int open_tcp();

// net_udp.cppm
export module net:udp;
export int open_udp();
```

Partition implementations

```
// net_tcp_impl.cpp
module net:tcp;
int open_tcp() { return 1; }

// net_udp_impl.cpp
module net:udp;
int open_udp() { return 2; }
```

Consumer

```
// app.cpp
import net;

int main() {
    return open_tcp() + open_udp();
}
```

This volume will show how to apply partitions responsibly. Partitions can be used to scale architecture, but they can also be abused to hide poor boundaries. We will treat partitioning as an architectural tool, not a dumping mechanism.

A staged migration model used in this volume

Large header-based systems should not be modularized by rewriting everything at once. Instead, this volume teaches a staged plan that preserves delivery:

- **Stage 1: Architectural discovery.** Identify boundaries, ownership, dependency direction, and the hotspots that cause build pain.
- **Stage 2: Create module-friendly seams.** Reduce macro leakage, isolate platform glue, and pull internal utilities behind narrow facades.
- **Stage 3: Introduce named modules for stable components.** Start where boundaries are already strong: domain core, math/algorithms, utilities with clean APIs.
- **Stage 4: Use header units tactically.** Import legacy headers where it reduces churn and speeds adoption, while gradually shrinking the global include surface.
- **Stage 5: Refactor away cycles and enforce ownership.** Turn cyclic include webs into explicit import graphs, then redesign responsibilities until cycles vanish.

- **Stage 6: Consolidate and harden.** Stabilize exports, validate build determinism, and establish long-term rules for new code.

This model is repeated in the "Full Project" part of the volume, where decisions are shown step-by-step, including trade-offs and the reasoning behind boundary choices.

Conventions and guarantees for examples

Throughout this volume:

- examples are written to emphasize **boundary design** and **export discipline**,
- internal details are kept internal intentionally,
- naming reflects architecture: `domain.*`, `core.*`, `util.*`, `platform.*`,
- code favors clarity and correctness over cleverness,
- patterns scale from small examples to multi-module layouts.

When build-system snippets are shown, they are structured around the modern requirement that module dependency order must be derived from **imports**, not from manual compilation ordering. Where toolchains differ, the volume presents a principled approach rather than fragile command-line recipes.

What you should expect by the end

By the final chapters, you should be able to look at a complex C++ system and confidently answer:

- What are the real components, and where are the false boundaries?

- What is the intended dependency direction, and where is it violated?
- Which interfaces are stable exports, and which are leaking internals?
- Where are cyclic dependencies coming from, and how do we remove them by redesign?
- How do we structure modules so builds become deterministic and scalable?
- How do we migrate without stopping delivery?

Modules are not a promise of magic. They are a promise of **structure**. This volume exists to ensure you can fulfill that promise in production-quality C++ systems, from C++20 to C++26, with engineering discipline, architectural clarity, and long-term maintainability as the primary goals.

Part I

Program Architecture & Modern Code Organization

Anatomy of a Modern C++ Program

1.1 High-level program architecture

1.1.1 High-level architectural goals for modern C++

A modern C++ program is successful long-term when its architecture achieves these properties simultaneously:

- **Explicit boundaries**: dependencies flow in a controlled direction; APIs are intentional.
- **Stable interfaces**: most changes remain local; rebuilds are contained.
- **Replaceable infrastructure**: platform/DB/network can evolve without rewriting the domain.
- **Observable behavior**: logging, tracing, metrics are designed-in, not added later.
- **Build scalability**: parallel compilation is effective; dependency graphs are predictable.
- **Portability**: source encoding and text handling work across OS/toolchains.

1.1.2 Application layers

Layering is the simplest reliable way to make large C++ systems comprehensible and evolvable. A practical layering model that maps well to libraries/targets and (later) modules is:

1. **Presentation / Interface Layer**
    - CLI commands, GUI views, HTTP endpoints, gRPC handlers, scripting adapters.
    - Main responsibilities: input validation, authentication/authorization checks, request/response mapping.
    - Avoid: business logic, persistence logic, and deep threading decisions.
2. **Application Layer**
    - Use-case orchestration: “place order”, “register user”, “process invoice”.
    - Coordinates domain objects, policies, repositories, and messaging.
    - Owns transactional workflow semantics (unit-of-work boundaries).
    - Defines application-level DTOs (data transfer objects) and mapping rules.
3. **Domain Layer**
    - Business rules, invariants, domain services, policies, domain events.
    - Prefers pure logic with minimal dependency on OS/frameworks.
    - Should be testable in isolation with deterministic behavior.
4. **Infrastructure Layer**
    - DB adapters, network stacks, OS integration, file I/O, TLS, timers, thread pools, codecs.
    - Implements application/domain-defined interfaces (repositories, gateways).
    - Integrates observability sinks (log backends, trace exporters).

Dependency rule (the rule that matters most):

- Domain does not depend on infrastructure.

- Application depends on domain and on *abstractions* of infrastructure, not concrete drivers.
- Infrastructure depends on domain/application abstractions to satisfy them.

Enforcing layer boundaries in C++

- **Targets**: one library per layer (or per bounded context); enforce a strict target dependency DAG.
- **Public vs private headers/modules**:
  - exported module interfaces or public headers define stable API;
  - implementation stays in `.cpp` or module implementation units.
- **Visibility discipline**:
  - prohibit including infrastructure headers in domain targets;
  - prohibit cross-feature includes unless explicitly approved.

1.1.3 Core library layer

The **core library layer** is the foundation other parts of the program treat as stable. It is not "utilities". It is the **core contract** of the program.

Typical contents of the core layer

- **Types**: strong IDs, value objects, domain primitives, small algebraic types.
- **Error model**: error codes/categories, exception policy, result/expected style types.
- **Resource model**: RAII handles, non-owning views, lifetime-safe adapters.
- **Configuration model**: validated configuration objects and parsing boundaries.

- **Time and units**: duration/timepoint wrappers, conversions, monotonic clock usage guidelines.
- **Core contracts**: repository interfaces, gateway interfaces, abstract clock/randomness providers.

Core layer constraints

- no network/DB/UI dependencies,
- minimal OS dependencies (prefer abstractions),
- minimal heavy headers and minimal template bloat in the exported surface,
- stable ABI boundaries where you ship binary components.

1.1.4 Utilities & abstractions

A healthy architecture separates **helpers** from **contracts**.

Utilities (helpers)

- tiny functions, adapters, formatting helpers, small algorithmic glue;
- should not carry program-wide policy decisions;
- must remain small, cohesive, and dependency-light.

Abstractions (contracts)

- stable interfaces that other layers depend on;
- should express meaning, not implementation.

Anti-pattern: the "util" dumping ground

If `util` imports/includes everything, it becomes a transitive dependency accelerator and makes builds collapse. A strict rule that prevents this:

> A utility header/module must not include/import application infrastructure unless it *belongs* to that infrastructure layer.

1.1.5 Domain-driven program structure

A domain-driven structure groups code by **business capability** (bounded contexts) rather than by technical type.

Feature-first folder layout

```
src/
  domain/
    billing/
      billing.model.cppm
      billing.policy.cpp
      billing.events.cppm
    identity/
      identity.model.cppm
      identity.auth.cpp
  app/
    billing/
      billing.usecases.cpp
    identity/
      identity.usecases.cpp
  infra/
```

```
    db/
    net/
    os/
tests/
```

Benefits

- reduces cross-feature coupling and shared-state temptation,
- aligns ownership and review boundaries with business capabilities,
- enables incremental modularization (one module per feature, then refine partitions).

Feature-level API pattern (domain)

```
// domain/billing/billing.model.cppm
export module acme.domain.billing;

export namespace acme::domain::billing {

  export struct InvoiceId {
    std::uint64_t value{};
  };

  export struct Money {
    std::int64_t cents{};
    constexpr Money operator+(Money o) const noexcept { return {cents + o.cents}; }
  };

  export struct Invoice {
    InvoiceId id{};
    Money total{};
  };
```

```
  export bool is_valid(const Invoice&) noexcept;
}
```

```
// domain/billing/billing.policy.cpp
module acme.domain.billing;

namespace acme::domain::billing {
  bool is_valid(const Invoice& inv) noexcept {
    return inv.id.value != 0 && inv.total.cents >= 0;
  }
}
```

1.2 Translation units, boundaries, and interfaces

1.2.1 Why translation units still define architecture

In C++, a file is not merely storage; it is a **compilation and linkage boundary**. Architecture must respect:

- **what gets compiled together** (translation units),
- **what becomes reachable/visible** (headers vs modules),
- **what becomes linkable and ODR-relevant**.

1.2.2 File-level interfaces

A file-level interface is the smallest enforceable unit of ownership. You should design it as a contract with these attributes:

- **Minimal surface area**: only declarations intended for consumers.

- **Minimal dependencies**: no heavy includes/imports in public API.
- **Clear lifetime rules**: ownership, borrowing, and thread-safety are explicit.
- **Clear error rules**: exceptions allowed or not; error channels defined.

Three interface patterns

1. **Header + source**: stable header declares; `.cpp` defines.
2. **Named module**: `.cppm` exports; implementation in module impl units.
3. **Facade**: small top-level API that delegates to internal partitions/headers.

Facade interface example (module)

```
// app/api/app.api.cppm
export module acme.app.api;

export namespace acme::app {
  export struct StartOptions {
    int port{};
  };

  export int run(StartOptions);
}
```

```
// app/api/app.api.impl.cpp
module acme.app.api;

import acme.app.runtime;     // internal module
import acme.infra.net;       // infrastructure module

namespace acme::app {
```

```
  int run(StartOptions opt) {
    return runtime::start_http_server(opt.port);
  }
}
```

1.2.3 TU explosion problem

TU explosion happens when small changes cause massive rebuilds and slow iteration. Architectural causes are usually visible in your dependency graph:

- deep include chains,
- large public headers with many transitive includes,
- header-only designs used everywhere,
- template-heavy APIs exposed widely without boundaries.

Symptoms

- touching a single header triggers recompiling “everything”,
- CI build time scales poorly with code size,
- developers fear refactoring because it breaks builds widely,
- ODR/link errors appear unpredictably due to macro-dependent builds.

Mitigation toolkit (header era and module era)

1. **Public API minimization**:
    - move heavy includes into implementation units,
    - forward declare when safe,

- avoid leaking third-party types in public interfaces.

2. **PImpl / type erasure where justified**:
   - reduce header dependency fan-out,
   - improve ABI stability for binary distribution.
3. **Explicit instantiation** for selected templates:
   - keep template definitions internal,
   - instantiate common combinations in implementation units.
4. **Modules for stable layers**:
   - export only what you truly want as API,
   - keep imports of heavy dependencies out of the primary interface (use partitions/impl units).

1.2.4 The cost of including headers

Textual inclusion is powerful but has structural costs:

- repeated parsing in every translation unit,
- macro leakage and configuration fragility,
- include-order sensitivity,
- accidental transitive coupling (a header pulls in a world of dependencies).

A strict “public header budget”

Define a policy for public interfaces (public headers or exported module interfaces):

- include/import only what is required to compile the declarations,
- never include platform headers in domain APIs,
- never include third-party headers in core APIs (wrap behind abstractions),
- never use `using namespace` in public interfaces.

Technique: dependency inversion by types

Instead of exposing a third-party type:

```
// BAD: public API leaks third-party type
// export SomeThirdPartyType parse(std::string_view);
```

Expose your own stable type:

```
// GOOD: stable domain type
export module acme.core.text;

export namespace acme::core {
  export struct ParsedText {
    // stable representation for your domain
    std::string utf8;
  };

  export ParsedText parse(std::string_view utf8_input);
}
```

1.2.5 ODR and its effect on architecture

The **One Definition Rule** influences architecture because it constrains how definitions can appear across translation units. In practice, architecture interacts with ODR through:

- inline functions and inline variables in headers,
- templates and their instantiations,
- macro-controlled declarations/definitions,
- "header-only" libraries and unity-build strategies.

ODR risk patterns

1. **Macro-dependent signatures**:
    - the same header produces different class layouts or function signatures in different TUs.
2. **Inline variables with differing initializers**:
    - especially when macros/flags change constants across TUs.
3. **Templates with hidden configuration**:
    - template behavior changes under macros without being visible in the type system.

Architecture rules that reduce ODR hazards

- keep macro usage out of exported/public APIs,
- isolate configuration into **build definitions** and **explicit config objects**,
- keep inline code small and stable,

- avoid exposing “configuration by include order” (fragile).

Explicit instantiation pattern (build-time and ODR control)

```
// public header / exported interface
#pragma once
#include <vector>

namespace acme {
  template<class T>
  struct Buffer {
    std::vector<T> v;
    void push(const T& x) { v.push_back(x); } // small inline ok
  };

  extern template struct Buffer<int>;  // declaration of explicit instantiation
}
```

```
// one .cpp in the library
#include "buffer.hpp"
namespace acme {
  template struct Buffer<int>; // explicit instantiation definition
}
```

This reduces repeated instantiations and can improve link-time determinism.

1.3 Namespaces as architectural tools

1.3.1 Namespaces as modules (before real modules)

Before C++20 modules, namespaces were the main language-level tool to partition the symbol space. They remain important even with modules:

- **Modules** control compilation boundaries and reachability.
- **Namespaces** control naming, discoverability, and conceptual grouping.

Layer-oriented namespace scheme

```
namespace acme::domain { /* pure domain */ }
namespace acme::app    { /* use-cases */ }
namespace acme::infra  { /* db/net/os adapters */ }
namespace acme::core   { /* shared primitives */ }
```

This communicates ownership instantly, improves grep-ability, and helps avoid dependency confusion.

1.3.2 Versioning APIs with namespaces

Namespace-based versioning is a practical technique to evolve APIs while preserving compatibility paths.

Version namespace with stable alias

```
namespace acme::api::v1 {
  struct Config { int mode{}; };
  int run(const Config&);
}

// Optional: stable alias for ``current''
namespace acme::api {
  using namespace v1;
}
```

Guidelines:

- version namespaces should be used for **public** APIs, not internal code,

- do not treat version namespaces as a substitute for ABI strategy; they complement it.

Inline namespaces (advanced) and the ABI warning

```
namespace acme::api {
inline namespace v1 {
  struct Token { std::uint64_t value{}; };
}
}
```

Inline namespaces can simplify source compatibility, but they can also hide changes. Use them only when you have a deliberate versioning plan and test coverage that detects ABI breaks.

1.3.3 Encapsulation strategies

`detail` namespace for non-API entities

```
namespace acme {
  namespace detail {
    inline std::uint64_t mix(std::uint64_t x) noexcept {
      x ^= x >> 33; x *= 0xff51afd7ed558ccdULL;
      x ^= x >> 33; x *= 0xc4ceb9fe1a85ec53ULL;
      x ^= x >> 33; return x;
    }
  }

  inline std::uint64_t stable_hash(std::uint64_t x) noexcept {
    return detail::mix(x);
  }
}
```

Rule: `detail` is not API. Never document it as stable.

TU-local encapsulation with unnamed namespace

```
namespace {
  int g_counter = 0;
  void bump() noexcept { ++g_counter; }
}

int next_value() noexcept {
  bump();
  return g_counter;
}
```

This confines state and helper functions to one translation unit.

Encapsulation by module reachability (preferred in C++20+)

```
// interface
export module acme.core.crypto;
export namespace acme::crypto {
  export std::uint64_t hash64(std::uint64_t) noexcept;
}
```

```
// implementation: helper is not exported and not reachable by importers
module acme.core.crypto;

namespace {
  std::uint64_t mix(std::uint64_t x) noexcept { /* ... */ return x; }
}

namespace acme::crypto {
  std::uint64_t hash64(std::uint64_t x) noexcept { return mix(x); }
}
```

1.3.4 Pitfalls in namespace usage

Namespaces can harm architecture if misused.

Pitfall: `using namespace` in public interfaces

- introduces invisible name pollution for every consumer,
- increases accidental ambiguity when dependencies grow.

Pitfall: too-deep nesting

Very deep namespaces harm readability:

```
// Hard to read and refactor:
namespace acme::company::division::project::module::subsystem::detail { }
```

Prefer meaningful boundaries with moderate depth:

```
namespace acme::domain::billing { }
namespace acme::infra::db { }
```

Pitfall: generic namespace buckets

Avoid global buckets like: `common`, `helpers`, `utils`. They become dependency magnets.

Pitfall: treating namespaces as access control

Namespaces are not access control. They reduce collisions and improve organization, but they do not enforce reachability the way modules do.

1.4 UTF-8 and source portability

1.4.1 Cross-platform encoding pitfalls

Source portability is not guaranteed by syntax alone. It depends on:

- **how source files are encoded** (UTF-8, UTF-8 with BOM, legacy code pages),
- **compiler assumptions** about the source character set,
- **execution character set** for narrow literals and runtime behavior,
- **OS boundary APIs** (UTF-16 wide APIs on Windows are common; UTF-8 is common on Unix-like systems).

Common failure modes:

- a file saved as UTF-8 is compiled as a legacy code page, corrupting string literals,
- differing defaults across compilers cause tests to pass on one OS and fail on another,
- console output/input depends on locale and terminal encoding,
- filesystem paths and user input contain Unicode and break naive `std::string` assumptions.

Repository-level policy

Adopt these repository policies:

1. All source files are UTF-8 (prefer without BOM unless your toolchain requires BOM).
2. Enforce consistent line endings via tooling.
3. Define compiler flags that force UTF-8 assumptions where applicable.
4. Add CI tests that include non-ASCII literals and filenames.

1.4.2 UTF-8, UTF-16, UTF-32 interactions

Unicode encodings are different representations of code points:

- UTF-8: variable-length, byte-oriented, excellent for storage/protocols.
- UTF-16: variable-length in 16-bit code units, common in Windows APIs.
- UTF-32: fixed-width in 32-bit code units, convenient for indexing by code point but memory heavy.

Architecture rule: "UTF-8 inside, convert at edges"

A high-leverage rule that scales:

- keep the **domain/core** using UTF-8 for text interchange,
- convert to platform-native representation at the boundary:
  - Windows: convert to UTF-16 when calling wide Win32 APIs,
  - Unix-like: UTF-8 often passes through to OS interfaces directly (still validate).
- never scatter conversions throughout domain logic.

The three distinct meanings you must not confuse

- **Bytes**: raw data buffer; may not be text.
- **Code units**: elements of an encoding (UTF-8 bytes, UTF-16 16-bit units).
- **Code points / grapheme clusters**: what humans perceive as characters (more complex than code points).

Portability failures often come from assuming:

one byte equals one character

which is false for UTF-8.

1.4.3 `char8_t`: new best practices

Modern C++ distinguishes UTF-8 string literals more explicitly. The `u8""` literal uses a UTF-8-oriented element type, and `std::u8string` / `std::u8string_view` represent UTF-8 code units.

Design guideline: make text-vs-bytes explicit

Adopt a strict meaning:

- `std::string` / `std::string_view` is **bytes or unspecified encoding**.
- `std::u8string` / `std::u8string_view` is **UTF-8 text**.

UTF-8-first API (recommended for text)

```
export module acme.core.text;

import <string>;
import <string_view>;
import <cstddef>;

export namespace acme::text {

  export using u8sv = std::u8string_view;

  export struct Utf8 {
    std::u8string s;
  };
```

```
  export Utf8 concat(u8sv a, u8sv b) {
    Utf8 out;
    out.s.reserve(a.size() + b.size());
    out.s.append(a);
    out.s.append(b);
    return out;
  }
}
```

Byte-view bridge (explicit and safe as bytes)

```
#include <string_view>
#include <cstddef>

inline std::string_view as_bytes(std::u8string_view s) noexcept {
  return { reinterpret_cast<const char*>(s.data()),
           static_cast<std::size_t>(s.size()) };
}
```

This conversion is appropriate when:

- you are handing UTF-8 bytes to a system that expects bytes,
- you are hashing, writing to a file, or sending over the network.

Do not use it to pretend you now have a *text* string in `std::string` with the same semantic guarantees. Keep the semantic boundary explicit.

Avoiding accidental overload confusion

Provide overload sets that accept both byte strings and UTF-8 text, but ensure each overload has explicit meaning:

```
namespace acme {

  void log_bytes(std::string_view bytes);        // opaque bytes
  void log_text(std::u8string_view utf8_text);   // UTF-8 semantics

}
```

UTF-8 literals in interfaces

```
constexpr std::u8string_view tag_core = u8"core";
constexpr std::u8string_view tag_net  = u8"net";
```

This avoids surprises across compilers and makes the intent unambiguous.

1.4.4 C++ source portability guidelines

A portable modern C++ program should establish and enforce these guidelines:

1) Source file encoding

- Save all `.cpp`, `.hpp`, `.cppm` as UTF-8.
- Decide on BOM usage and enforce consistently.
- Validate encoding in CI (a simple script can reject non-UTF-8 files).

2) Compiler character set policy

- Ensure the compiler treats source as UTF-8 consistently across platforms.
- Ensure execution character set expectations are consistent for narrow literals.
- Prefer avoiding locale-dependent behavior for correctness.

3) Text policy inside the program

- Choose a single internal interchange encoding for text (commonly UTF-8).
- Keep conversions at boundaries (OS APIs, UI frameworks, filesystem adapters).
- Avoid mixing text encodings within a single layer.

4) Filesystem and user input

- Treat filesystem paths as a boundary concern.
- Add tests for Unicode filenames and user input.
- Do not assume console encoding; treat it as presentation.

5) Testing strategy

- Unit tests for UTF-8 parsing/validation/normalization policies (if relevant).
- Integration tests that validate end-to-end Unicode handling:
  - file I/O with Unicode filenames,
  - network payloads with UTF-8 JSON,
  - UI/CLI output correctness where applicable.

6) "Definition of done" for portability

Declare portability complete only when:

- code compiles cleanly on at least two major toolchains,
- tests run successfully on at least two OS families,
- Unicode edge cases are exercised in CI, not only locally.

Part II

Modern C++ Modules: Architecture, Design, Engineering

The Philosophy of Modules

2.1 The historical header problem

2.1.1 Why the include model became a scaling limit

C and C++ grew around a compilation model in which interfaces are delivered by **textual inclusion**: a translation unit is formed by running the preprocessor, expanding macros, and effectively pasting included headers into each `.cpp` file. This model was powerful in the 1980s and 1990s because it was simple, portable, and toolchain-friendly. However, the very properties that made it successful also create structural limits at scale:

- **Textual inclusion is not a dependency boundary**: it is text substitution.
- **The preprocessor is not namespaced**: it is global-state mutation.
- **Headers encourage transitive coupling**: one include pulls many others.
- **The same header is parsed repeatedly**: once per translation unit, often hundreds or thousands of times per build.
- **Build correctness depends on configuration consistency**: macros, include paths, and compiler flags must match across all users.

These issues manifest as macro pollution, fragile inheritance and ABI patterns, slow builds, and architectural erosion in very large codebases.

2.1.2 Macro pollution

The preprocessor operates **before** the C++ language rules. It has no scope rules, no namespaces, and no type checking. As a result, macro definitions and macro-controlled code in headers behave like **ambient global state**. This creates three practical hazards:

1. **Name collisions**: a macro can hijack an identifier.
2. **Semantic instability**: the same header can produce different declarations in different translation units depending on macro state.
3. **Configuration leakage**: a macro intended for one library affects unrelated code that includes the same headers later.

Example 1: name collision that silently changes meaning

```
// legacy_macros.hpp
#pragma once
#define min(a,b) ((a) < (b) ? (a) : (b))

// app.cpp
#include "legacy_macros.hpp"
#include <algorithm>

int f(int x, int y) {
  // Intended: std::min(x, y)
  // Reality: macro expansion can break the call or change meaning.
  return std::min(x, y);
}
```

This is not “just style”; it is a fundamental property: macros are not confined to a library boundary.

Example 2: macro-controlled API shape causes cross-TU inconsistency

```
// config_api.hpp
#pragma once

#ifdef USE_64BIT_ID
using id_type = std::uint64_t;
#else
using id_type = std::uint32_t;
#endif

struct Record {
  id_type id;
};
```

```
// a.cpp
#define USE_64BIT_ID
#include "config_api.hpp"
Record make_a();

// b.cpp
// does NOT define USE_64BIT_ID
#include "config_api.hpp"
Record make_b();
```

Now `Record` is **not the same type** across translation units. This leads to ODR violations, ABI mismatches, and undefined behavior at link/runtime.

Architectural rule: macros must not define public contracts

In modern large-scale C++:

- macros may exist for platform detection and compatibility shims,
- but public API shape must not depend on uncontrolled macro state,
- configuration should be expressed through build-system options and explicit runtime/compile-time config objects,
- public headers must remain robust under arbitrary include order.

2.1.3 Fragile base class problem

The “fragile base class” problem is traditionally explained as an object-oriented design risk: changes to a base class can unexpectedly break derived classes. In C++, headers amplify this problem because inheritance relationships and virtual layouts are **distributed** through textual inclusion.

Why C++ makes base classes especially sensitive

- Virtual functions introduce vtables and object layout constraints.
- Adding a virtual function, data member, or changing access can affect derived classes and binary compatibility.
- Inline member function definitions in headers propagate changes into every translation unit.

Example: a seemingly small base change triggers a rebuild avalanche

```
// base.hpp
#pragma once
#include <string>

struct Base {
```

```
  virtual ~Base() = default;
  virtual std::string name() const = 0;

  // Adding this inline helper forces recompilation of many TUs:
  std::string debug() const { return "Base(" + name() + ")"; }
};
```

Even if the new function does not change the vtable (it is non-virtual), placing it inline in the header causes:

- a mass rebuild (because the header changed),
- potential performance and code size effects if the compiler inlines it everywhere,
- a new transitive dependency footprint if it pulls in more headers (here: `<string>` already exists, but in real systems it grows).

Architecture-level mitigations

- Prefer stable interfaces: non-virtual interfaces, composition, and narrow extension points.
- Keep base classes minimal; avoid inline non-trivial member bodies in public headers.
- Use PImpl or type-erasure when ABI stability and build containment are priorities.
- In a module world, prefer exporting a clean interface and keeping implementation details in implementation units.

2.1.4 Slow builds

Header-based builds are slow primarily because they do the same work repeatedly. For each translation unit, the toolchain:

- preprocesses and expands macros,

- parses all included headers,
- performs semantic analysis and instantiates templates (often repeatedly),
- generates code for the TU,
- then repeats the whole process for the next TU.

The common "optimization" strategies (precompiled headers, unity builds) help, but they do not solve the underlying architectural issue:

- PCH reduces repeated parsing but can be fragile and configuration-sensitive.
- Unity builds reduce overhead but can increase coupling, change ODR behavior, and hide missing includes.

The hidden cost: include depth and transitive dependencies

Many codebases slow down not because a file includes *one* heavy header, but because the include graph becomes deep and highly connected:

- one public header includes another public header,
- which includes platform adapters,
- which include third-party libraries,
- producing large parse sets everywhere.

2.1.5 Unscalable large codebases

At small scale, headers feel natural. At large scale, they create systemic problems:

1. **Hidden coupling**: dependency edges are created implicitly by includes.

2. **Unclear ownership**: any file can include any header, making layering rules advisory rather than enforceable.
3. **Refactoring fear**: changing a core header triggers rebuilds and can break many consumers.
4. **Configuration hazards**: different macro states across components cause subtle ABI/ODR faults.

These are not merely build issues; they are architectural issues: the include model blurs the line between what is **API** and what is **implementation detail**.

2.2 Motivation behind C++20 modules

2.2.1 The core idea: imports replace textual inclusion

C++20 modules introduce a fundamentally different compilation relationship:

- `#include`: paste text into the current translation unit (with all macro/preprocessor consequences).
- `import`: depend on a separately compiled interface (a compiled representation of declarations).

This shifts the model from "compile by textual assembly" to "compile by explicit interface dependencies".

Include-based interface

```
// math.hpp
#pragma once
```

```
int add(int, int);

// app.cpp
#include "math.hpp"
```

Module-based interface

```
// math.cppm
export module acme.math;
export int add(int, int);

// app.cpp
import acme.math;
```

The module interface is compiled once (per configuration) and then imported, rather than re-parsed as text in every consumer.

2.2.2 Import/export vs include

The **export/import** model changes three key aspects of program design:

1) API becomes explicit and curated

Only declarations explicitly marked for export become part of the module's public surface.

```
// acme.crypto.cppm
export module acme.crypto;

export namespace acme::crypto {
  export std::uint64_t hash64(std::uint64_t) noexcept;
}

// Not exported: not part of API
```

```
namespace acme::crypto::detail {
  std::uint64_t mix(std::uint64_t) noexcept;
}
```

Consumers import a clean API surface; internal details are not accidentally reachable.

2) Preprocessor leakage is reduced

Modules are designed so that importing a module does not behave like pasting its source text into the importer. As a result, the typical "macro pollution" failure mode is dramatically reduced.

3) Dependency boundaries become tool-enforceable

Because imports are explicit, build systems can scan and order compilation steps correctly, and large projects can formalize dependency DAGs more reliably.

2.2.3 Logical vs physical code boundaries

A long-standing mismatch in header-based C++ is that:

- the **logical boundary** (library API vs implementation) is conceptual,
- but the **physical boundary** (what the compiler sees) is blurred by textual inclusion.

Modules realign these:

- the module interface unit is the physical artifact representing the public interface,
- implementation units are physical artifacts that remain private to the module,
- consumers interact with the module at the logical level via `import`.

Design consequence: “public means exported”

In a module-first architecture:

- you design the interface unit like a public contract document,
- you treat implementation units as internal evolution space,
- you avoid exporting types that expose heavy dependencies or unstable details.

2.2.4 Compilation performance gains

The performance promise of modules comes from reducing redundant work:

- module interfaces are compiled once into a compiled representation,
- importers reuse that representation rather than re-parsing the same headers repeatedly,
- dependencies become explicit, enabling better build scheduling and parallelism.

However, realistic guidance for engineers is:

- compilation gains depend on toolchain maturity and project structure,
- the biggest wins typically occur when large, stable interfaces are imported widely,
- naive modularization (exporting everything, importing everything) can reduce gains.

Example: “fat interface” vs partitioned interface

A monolithic interface that imports everything will still be expensive to compile and to reuse:

```
// BAD: monolithic interface pulls many heavy dependencies
export module acme.all;
```

```
import <vector>;
import <string>;
import <map>;
import <filesystem>; // heavy
import acme.infra.net; // heavy, and shouldn't be exported to everyone

export namespace acme {
  export void do_everything();
}
```

A better design keeps the primary interface small and uses partitions/implementation units:

```
// GOOD: small facade + exported partitions
export module acme.core;

export import :types;
export import :algo;

export namespace acme {
  export int add(int, int);
}
```

```
// acme.core:types.cppm
export module acme.core:types;

import <string>;
export namespace acme {
  export struct Name { std::string utf8; };
}
```

```
// acme.core:algo.cppm
export module acme.core:algo;

import <vector>;
```

```
export namespace acme {
  export int sum(const std::vector<int>&);
}
```

This structure improves:

- parallel compilation of partitions,
- containment of heavy imports,
- the stability of what most consumers need to import.

2.2.5 Encapsulation improvements

Encapsulation is the philosophical heart of modules. Headers can express intent, but they cannot enforce reachability in the same way because inclusion is textual.

Encapsulation in header-based design: conventions

In header-based design, we rely on conventions:

- `detail/` directories,
- `internal.hpp` naming,
- comments like “not part of the public API”,
- private include paths not exposed to consumers.

These conventions work until they do not: a consumer can still include an internal header if it can find it.

Encapsulation in module-based design: reachability

With named modules:

- only exported declarations are reachable by importers,
- internal declarations in implementation units are not reachable,
- macro and include-order hazards are reduced because import is not textual inclusion.

Example: hiding implementation details without PImpl

```
// interface: acme.storage.cppm
export module acme.storage;

export namespace acme::storage {
  export struct Key { std::uint64_t v{}; };

  export class Store {
  public:
    Store();
    ~Store();

    Store(Store&&) noexcept;
    Store& operator=(Store&&) noexcept;

    Store(const Store&) = delete;
    Store& operator=(const Store&) = delete;

    void put(Key, std::string_view utf8);
    std::string get(Key) const;

  private:
    struct Impl;     // declared but not exported as a definition
```

```
    Impl* p_;        // opaque pointer (still possible), but implementation can be fully
    ↪  private in module impl
  };
}
```

```
// implementation: acme.storage.impl.cpp
module acme.storage;

import <unordered_map>;
import <string>;
import <string_view>;

namespace acme::storage {

  struct Store::Impl {
    std::unordered_map<std::uint64_t, std::string> data;
  };

  Store::Store() : p_(new Impl{}) {}
  Store::~Store() { delete p_; }

  Store::Store(Store&& o) noexcept : p_(o.p_) { o.p_ = nullptr; }
  Store& Store::operator=(Store&& o) noexcept {
    if (this != &o) { delete p_; p_ = o.p_; o.p_ = nullptr; }
    return *this;
  }

  void Store::put(Key k, std::string_view utf8) {
    p_->data[k.v] = std::string(utf8);
  }

  std::string Store::get(Key k) const {
    auto it = p_->data.find(k.v);
    return (it == p_->data.end()) ? std::string{} : it->second;
```

```
  }
}
```

Even here, modules improve encapsulation because:

- consumers cannot reach `Store::Impl` because its definition is not exported,
- heavy containers and headers are imported only in the implementation unit,
- build dependencies become cleaner and rebuild impact is reduced.

2.2.6 A realistic engineering view: modules require build-system cooperation

The philosophy of modules is not purely language-level. Modules introduce a build-order requirement:

- importers require compiled module interfaces to exist before compilation,
- build systems must discover and order module dependencies,
- toolchains produce compiled interface artifacts (often called BMI; MSVC uses IFC terminology for its compiled interfaces).

Therefore, adopting modules is both an architectural decision and a build engineering decision.

2.2.7 Practical summary: what modules are trying to fix

C++20 modules are motivated by a consistent set of goals:

1. Replace textual inclusion with explicit import of compiled interfaces.
2. Reduce macro and include-order fragility across library boundaries.
3. Enable scalable builds by avoiding repeated parsing of large headers.

4. Improve encapsulation by making reachability depend on exported surfaces rather than conventions.

5. Provide a foundation for large-scale program organization where the physical compilation model matches logical architecture.

2.3 Exercises

2.3.1 Exercise 1: Demonstrate macro pollution and contain it

1. Create a header that defines a macro named `min` or `check`.

2. Include it before `<algorithm>` and observe failures or surprising behavior.

3. Refactor so that the macro is confined to a `.cpp` file and verify that the public interface is macro-free.

2.3.2 Exercise 2: Refactor a small library to a module interface

1. Take a header-based library with a public API and an internal helper header.

2. Create a module interface exporting only the public API.

3. Move helpers to module implementation units.

4. Ensure consumers can no longer “reach” internal helpers by import alone.

2.3.3 Exercise 3: Partition a fat interface

1. Identify a header/module that pulls in many dependencies.

2. Split into an exported facade and one or more exported partitions.

3. Move heavy imports/includes into partitions or implementation units.

4. Measure whether rebuild impact and compile latency improve in your environment.

Module Syntax, Structure & Semantics

3.1 Declaring modules

3.1.1 From headers to module units: what the compiler actually sees

A **module unit** is a translation unit that contains a **module declaration**. Unlike header-based design (where a "library boundary" is a convention), a module unit establishes a real, language-level boundary:

- the module name becomes part of the program's interface graph,
- the exported declarations become the reachable interface for importers,
- non-exported declarations remain internal (subject to reachability rules),
- compilation requires the module interface to be processed before importers.

C++ recognizes two major module mechanisms:

- **Named modules**: your own modules such as `acme.core`, `acme.domain.billing`.
- **Header units**: importing a header (or a library-provided header unit) rather than textually including it.

3.1.2 Module interface units

A **module interface unit** is the primary place where a module publishes its API. It contains an **export module** declaration and then exports declarations.

Minimal module interface

```
// File: acme.math.cppm
export module acme.math;

export namespace acme::math {
  export int add(int a, int b);
  export int mul(int a, int b);
}
```

Key properties:

- Only entities made reachable via `export` become part of the importable interface.
- Implementation details can exist in the interface unit, but exporting them makes them API.
- Large or heavy dependencies should typically not be imported in the primary interface unless they are truly part of the public contract.

Export granularity and the “API budget” rule

Treat your module interface as a public contract document:

- export only what you commit to support,
- minimize the number of exported names,
- avoid exporting types that leak third-party dependencies,
- prefer exporting stable value types and narrow functions over exposing internal structures.

Interface unit with a curated exported surface

```
// File: acme.core.text.cppm
export module acme.core.text;

import <string>;
import <string_view>;

export namespace acme::core::text {
  export struct Utf8 {
    std::string bytes; // policy: UTF-8 encoded bytes
  };

  export Utf8 normalize(std::string_view utf8_input);
}
```

Here, `import <string>` is an implementation convenience for the interface itself. Whether you import `<string>` or rely on a standard library module such as `import std;` is a toolchain/build decision, but the architectural rule remains:

> Do not bloat the interface with heavy or unstable dependencies unless they are essential to the contract.

3.1.3 Implementation units

A **module implementation unit** belongs to a module and provides definitions not required to be exported. It begins with `module <name>;` (not `export module`).

Typical module implementation unit

```
// File: acme.math.impl.cpp
module acme.math;
```

```
namespace acme::math {
  int add(int a, int b) { return a + b; }
  int mul(int a, int b) { return a * b; }
}
```

Engineering implications:

- Changing an implementation unit should not force recompilation of importers, because importers depend on the compiled interface.
- Heavy includes/imports belong here (containers, filesystem, OS headers, third-party headers).
- This is where you concentrate non-exported helper functions, internal state, and heavy algorithms.

Internal helpers in an implementation unit

```
// File: acme.core.text.impl.cpp
module acme.core.text;

import <string>;
import <string_view>;

namespace {
  bool is_ascii(unsigned char c) noexcept { return c < 0x80; }
}

namespace acme::core::text {
  Utf8 normalize(std::string_view s) {
    // Example placeholder logic: real normalization may require a Unicode library.
    // The key point is architectural: heavy dependencies stay internal.
    Utf8 out;
```

```
    out.bytes.reserve(s.size());
    for (unsigned char c : s) {
      out.bytes.push_back(is_ascii(c) ? char(c) : char(c)); // placeholder
    }
    return out;
  }
}
```

3.1.4 Global module fragments

The **global module fragment** begins with a standalone `module;` at the start of a module unit. Its purpose is to allow **textual inclusion** (`#include`) and other preprocessing directives *before* the module declaration.

Why the global module fragment exists

- Migration: many libraries still require headers (platform headers, third-party headers, legacy C headers).
- Some headers must be included with particular macro configurations.
- It provides an explicit "this is still textual" zone separated from the module's exported interface.

Example: using a global module fragment for legacy headers

```
// File: acme.platform.win.cppm
module; // global module fragment begins

#define NOMINMAX
#include <windows.h>

export module acme.platform.win;
```

```
export namespace acme::platform::win {
  export int message_box(const wchar_t* text);
}
```

```
// File: acme.platform.win.impl.cpp
module acme.platform.win;

int acme::platform::win::message_box(const wchar_t* text) {
  return ::MessageBoxW(nullptr, text, L"acme", MB_OK);
}
```

Rules of thumb for the global module fragment:

- keep it minimal: every include there is potentially part of the compiled interface environment,
- avoid putting heavy headers there unless absolutely necessary,
- avoid leaking macro-based configuration into what becomes the module's public contract.

3.1.5 Private module fragments

A **private module fragment** begins with `module : private;` inside a *module interface unit*. Conceptually, it marks:

> The remainder of this interface unit is implementation detail and should not affect importers.

Single-file module pattern using a private module fragment

```
// File: acme.tiny.cppm
export module acme.tiny;
```

```
export int answer();

module : private;

int answer() { return 42; } // implementation detail in the private fragment
```

When to use a private module fragment:

- very small modules where splitting into multiple files is unnecessary,
- prototyping, educational modules, or stable micro-APIs,
- when you want a strict “interface then implementation” structure in one file.

When *not* to use it:

- large libraries with many contributors where one file becomes a choke point,
- cases where build tooling or compiler support for private fragments is known to be incomplete in your target toolchains,
- when you need multiple interface units (the module design assumes a single primary interface for a named module).

3.2 Importing modules

3.2.1 The import model: what import does and does not do

An **import declaration** introduces names from another module interface into the current translation unit. Crucially:

- **import is not textual inclusion**: it does not paste source code into the importer,

- it relies on a compiled representation of the module interface,
- it makes exported declarations available for name lookup according to module rules.

3.2.2 Standard library modules

Modern toolchains increasingly provide module-friendly access to the standard library. There are two major patterns you will see in practice:

1) Importing the standard library as a named module

```
// Example pattern (availability depends on toolchain and configuration)
import std;

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

2) Importing standard headers as header units

```
// Example pattern (availability depends on toolchain and configuration)
import <vector>;
import <iostream>;

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

Engineering guidance:

- Choose one strategy per build configuration (avoid chaotic mixing).

- If you import `std` (or a toolchain-provided standard library module), avoid also importing many individual standard headers as header units unless you have a deliberate reason.
- Treat the standard library import strategy as part of your build-system architecture, not as a local coding preference.

3.2.3 Import rules & visibility

The most important practical rules are:

Rule 1: module declaration position

In a module unit, the first meaningful declaration is governed by module rules:

- if you use a global module fragment, it begins with `module;` and contains preprocessing directives,
- the named module declaration (`export module X;` or `module X;`) comes next,
- imports typically appear after the module declaration in a clean, consistent order.

Rule 2: exported vs non-exported names

Only exported declarations are reachable by importers. Non-exported declarations may exist, but they are internal:

- export is the boundary between “library contract” and “private detail”,
- do not export internal helper templates, debug hooks, or experimental types unless you accept API obligations.

Rule 3: re-exporting through a facade

You can create an architectural facade module that re-exports partitions or other modules:

```
// File: acme.core.cppm
export module acme.core;

// Re-export selected partitions as the public umbrella
export import :types;
export import :algo;
```

```
// File: acme.core:types.cppm
export module acme.core:types;
import <string>;

export namespace acme::core {
  export struct Name { std::string utf8; };
}
```

```
// File: acme.core:algo.cppm
export module acme.core:algo;
import <vector>;

export namespace acme::core {
  export int sum(const std::vector<int>& v) {
    int s = 0;
    for (int x : v) s += x;
    return s;
  }
}
```

This facade pattern is a key architectural tool:

- most users import only the umbrella module,

- advanced users may import partitions directly if permitted,
- internal organization can evolve without forcing all consumers to learn your internal file layout.

3.2.4 Conditional imports

Conditional imports are an engineering need for portability and feature selection. While modules aim to reduce macro influence across boundaries, the preprocessor still exists, and build configurations still differ.

Pattern A: conditional import for toolchain/environment selection

```
// File: acme.platform.cppm
export module acme.platform;

#if defined(_WIN32)
export import acme.platform.win;
#else
export import acme.platform.posix;
#endif

export namespace acme::platform {
  export void init();
}
```

```
// File: acme.platform.impl.cpp
module acme.platform;

namespace acme::platform {
  void init() {
    // Platform-specific init is provided by imported submodule.
  }
```

```
}
```

Guidelines:

- keep conditional logic at a small number of boundary modules (platform facade),
- avoid scattering `#if` across many core modules,
- avoid using macros to change public API shapes; prefer selecting implementations.

Pattern B: conditional import of standard library module vs headers

In multi-toolchain environments, you may need a portability shim. Instead of having every file decide, centralize it:

```
// File: acme.std.cppm
export module acme.std;

// This file defines the policy for standard library access for the whole program.

#if defined(ACME_USE_STD_MODULE)
import std;            // named standard library module (toolchain-dependent)
#else
import <vector>;       // header units or regular includes depending on toolchain support
import <string>;
import <iostream>;
#endif

export namespace acme::stdshim {
  // No exports required; importing this module establishes the standard library policy.
  // Optionally export helper aliases/policies if you want a stable program-wide facade.
}
```

Then everywhere else:

```
import acme.std;
```

This prevents policy fragmentation and makes builds predictable.

3.2.5 Best practices for importing

The following practices scale well in real codebases:

Best practice 1: keep the primary interface light

- avoid importing heavy subsystems into the primary interface unless they are truly required by exported declarations,
- move heavy dependencies into partitions or implementation units.

Best practice 2: separate “API types” from “heavy algorithms”

Export small stable types from a `:types` partition, and keep heavy algorithms in `:algo` or implementation units. This reduces rebuild impact and improves comprehension.

Best practice 3: never mix “include-order contracts” with imports

A common legacy failure mode is relying on include order to configure macros. Modules reduce this fragility; do not reintroduce it by mixing includes and imports randomly. If you must include legacy headers, confine them to a global module fragment or implementation units.

Best practice 4: design for a stable dependency DAG

Avoid cycles in imports. If two components need each other, introduce an abstraction module (interfaces/types) that both depend on.

3.3 Module partitions

3.3.1 Why partitions exist

A named module has one primary interface, but real libraries are large. Partitions solve:

- **parallel development**: multiple contributors work on separate interface partitions,
- **build parallelism**: partitions can be compiled independently,
- **API organization**: stable grouping of exported declarations by topic.

3.3.2 Exported partitions

An **exported partition** is a module partition whose declarations are part of the module's public surface when re-exported.

Exported partition example

```
// File: acme.net:types.cppm
export module acme.net:types;

import <string>;
import <cstdint>;

export namespace acme::net {
  export struct Endpoint {
    std::string host;
    std::uint16_t port{};
  };
}
```

```
// File: acme.net:client.cppm
```

```
export module acme.net:client;

import <string_view>;
export import :types;

export namespace acme::net {
  export class Client {
  public:
    Client();
    void connect(const Endpoint&);
    void send(std::string_view utf8_payload);
  };
}
```

```
// File: acme.net.cppm (primary interface)
export module acme.net;

export import :types;
export import :client;
```

Here:

- `:types` is exported and becomes part of the public umbrella,
- `:client` exports its own declarations and reuses `:types`,
- the primary interface re-exports both partitions as the module's intended public surface.

3.3.3 Private partitions

A **private partition** exists for internal organization without becoming public API. A practical design pattern is:

- exported partitions contain only stable public types and functions,

- private partitions contain internal helpers, parsing logic, protocol codecs, and heavy algorithms,
- implementation units contain OS integration and third-party glue.

Private partition example (internal helpers)

```
// File: acme.net:codec.cppm (intended as internal)
// Note: This is not exported from the primary interface.
module acme.net:codec;

import <string>;
import <string_view>;

namespace acme::net::detail {
  std::string encode_frame(std::string_view payload) {
    std::string out;
    out.reserve(payload.size() + 2);
    out.push_back('[');
    out.append(payload);
    out.push_back(']');
    return out;
  }
}
```

```
// File: acme.net.client.impl.cpp
module acme.net:client;

import <string_view>;
import <iostream>;

import :codec; // imports private partition within the module

namespace acme::net {
```

```
  Client::Client() = default;

  void Client::connect(const Endpoint& ep) {
    std::cout << "connect " << ep.host << ":" << ep.port << "\n";
  }

  void Client::send(std::string_view utf8_payload) {
    auto frame = detail::encode_frame(utf8_payload);
    std::cout << "send " << frame << "\n";
  }
}
```

Notice:

- `:codec` is not exported from the primary interface, so importers of `acme.net` do not gain access to it,
- internal helpers remain internal without relying on folder naming conventions like `detail/`.

3.3.4 Partition dependency graphs

Partitions form a dependency graph inside a module. A scalable internal architecture follows these rules:

Rule 1: make one partition the “types” foundation

- `:types` exports stable value types and small enums.
- Most other partitions may depend on `:types`.
- `:types` should depend on as little as possible.

Rule 2: forbid cycles

Cyclic imports inside partitions create build-order complexity and often indicate a missing abstraction. When two partitions want each other:

- extract shared types into `:types` or `:interfaces`,
- or introduce a new internal partition that both import.

Rule 3: keep heavy dependencies away from the facade

If `acme.net` primary interface imports heavy partitions (filesystem, TLS stacks, JSON libraries), then every importer pays for it. Instead:

- export types and minimal facades,
- push heavy imports into implementation units or optional partitions.

A clean partition graph

```
acme.net (primary interface)
  exports :types
  exports :client
  exports :server

:client
  imports :types
  imports :codec (private)

:server
  imports :types
  imports :codec (private)
```

```
  imports :reactor (private)


:types
  imports only small standard components (string, cstdint)
```

A problematic partition graph (what to avoid)

```
:types imports :client
:client imports :types
```

This is a cycle; fix it by extracting shared contracts into `:types` and preventing `:types` from depending on higher-level partitions.

3.3.5 Designing partition hierarchies

A reliable hierarchy for large modules is:

Tier 1: exported foundational partitions

- `:types` (value types, ids, enums)
- `:errors` (error codes, categories, error contracts)
- `:api` (thin exported facades)

Tier 2: exported functional partitions

- `:client`, `:server`, `:algo`, `:format`
- each depends on tier 1
- minimal additional dependencies in their exported declarations

Tier 3: private/internal partitions

- `:detail_codec`, `:parser`, `:impl_support`
- heavy algorithms, parsing, internal tables
- not re-exported from the primary interface

Tier 4: implementation units

- OS integration, networking sockets, TLS backends, DB drivers
- bridges to third-party libraries
- platform-specific code and build configuration hooks

Example: hierarchy with a stable facade

```
// File: acme.image.cppm
export module acme.image;

// Export the stable public surface only:
export import :types;     // exported
export import :decode;    // exported facade
export import :encode;    // exported facade

// Internal partitions are not exported here:
// :png_impl, :jpeg_impl, :tables, etc.
```

```
// File: acme.image:types.cppm
export module acme.image:types;

import <cstdint>;
import <vector>;
```

```
export namespace acme::image {
  export struct Pixel { std::uint8_t r{}, g{}, b{}, a{255}; };

  export struct Image {
    std::uint32_t w{}, h{};
    std::vector<Pixel> pixels;
  };
}
```

```
// File: acme.image:decode.cppm
export module acme.image:decode;

export import :types;
import <string_view>;

export namespace acme::image {
  export Image decode(std::string_view bytes); // bytes, not text
}
```

```
// File: acme.image.decode.impl.cpp
module acme.image:decode;

import :png_impl;     // private/internal
import :jpeg_impl;    // private/internal
import <string_view>;

namespace acme::image {
  Image decode(std::string_view bytes) {
    // Choose based on signature; implementation hidden from importers.
    if (bytes.size() >= 8) {
      // placeholder
    }
    return Image{};
```

```
  }
}
```

This design keeps the exported API stable and small while allowing internal codecs and third-party integrations to evolve freely.

3.4 Practical checklist for Chapter 3

Use this checklist to review a module design:

1. **Interface discipline**: Are exported declarations minimal and stable?
2. **Dependency discipline**: Does the primary interface avoid heavy imports?
3. **Partition hierarchy**: Do you have a `:types` foundation and clear exported vs private partitions?
4. **No cycles**: Is the partition import graph acyclic?
5. **Centralized policy**: Are conditional imports centralized in a small number of facade modules?
6. **Edge conversions**: Are platform and third-party conversions kept in implementation units?

3.5 Exercises

3.5.1 Exercise 1: Convert a header library into interface + implementation units

1. Create `export module acme.demo;` and export a small API.

2. Move all heavy includes and implementations into `module acme.demo;` implementation units.

3. Verify that changes in implementation units do not force recompilation of importers (in your build environment).

3.5.2 Exercise 2: Introduce exported and private partitions

1. Create `:types` and export it from the primary interface.

2. Create a private partition `:codec` containing helper code and ensure it is not exported.

3. Use `import :codec;` only from inside module implementation units and confirm consumers cannot access internal helpers.

3.5.3 Exercise 3: Design a partition dependency graph

1. Draw a graph for your module partitions.

2. Ensure `:types` has minimal dependencies and no cycles exist.

3. Refactor until the graph is a clean DAG with a stable facade.

Module Engineering in Large Projects

4.1 Designing module-based architectures

4.1.1 API/ABI boundary decisions

When engineering large systems, the most important decision is not "how to write a module," but rather:

> Where does the API boundary live, and what are the ABI consequences?

Modules affect **compilation boundaries**, but they do not automatically define **binary boundaries**. Therefore, architects must explicitly distinguish:

- **Source-level API boundary**: what is exported from a module interface.
- **Binary-level ABI boundary**: what is visible across shared library or DLL boundaries.
- **Implementation boundary**: what remains internal and can change without affecting consumers.

Design rule: do not equate module = shared library

A module:

- defines compilation reachability,
- does not mandate binary linkage boundaries,
- may span multiple static or shared libraries,
- may exist entirely inside one binary.

Therefore, decide early:

1. Which modules are internal-only?
2. Which modules form external API contracts?
3. Which modules are stable ABI components?

Stable ABI module design

For modules crossing binary boundaries (DLL/shared library):

- Avoid exposing inline-heavy templates as ABI contracts.
- Avoid exposing STL containers directly if ABI stability across compiler versions matters.
- Prefer opaque handles, pImpl, or C-compatible boundary layers.
- Export stable value types with fixed layout.

Example: ABI-stable facade module

```
// acme.runtime.cppm
export module acme.runtime;

export namespace acme::runtime {
```

```
  export struct Config {
    int mode{};
  };

  export class Engine {
  public:
    Engine(Config);
    ~Engine();

    Engine(Engine&&) noexcept;
    Engine& operator=(Engine&&) noexcept;

    Engine(const Engine&) = delete;
    Engine& operator=(const Engine&) = delete;

    void start();
    void stop();

  private:
    struct Impl;
    Impl* p_; // stable opaque pointer
  };
}
```

This design:

- keeps implementation details out of the ABI surface,
- reduces binary breakage when internal structures evolve,
- aligns module boundary with ABI discipline.

4.1.2 Principled layering

Modules enable strong architectural layering because import graphs are explicit.
A principled layering strategy for large systems:

Layer 1: Foundational types

- Pure value types, identifiers, enums.
- No OS, no networking, no database.
- Minimal dependencies.

Layer 2: Core services

- Algorithms, validation, transformation logic.
- May depend on foundational types.

Layer 3: Adapters

- OS integration, networking, filesystem.
- Depend on layers 1 and 2.

Layer 4: Application entrypoints

- CLI, GUI, REST, batch drivers.
- Depend on all lower layers.

Architectural invariant

Imports must form a directed acyclic graph (DAG). No lower layer may import a higher layer.

4.1.3 Import cycles and how to avoid them

Import cycles are illegal in module dependency graphs. If two modules require each other, the design is incomplete.

Common cause of cycles

- Two modules define types that reference each other.
- Bidirectional ownership or control logic.
- Mixing interface definitions with implementation logic.

Refactoring pattern: extract shared contracts

```
// Instead of:

// module A imports B
// module B imports A

// Extract shared types:

export module acme.contracts;

export namespace acme {
  export struct Event { int id{}; };
}
```

Then:

- Module A imports `acme.contracts`.
- Module B imports `acme.contracts`.
- Neither imports the other.

Interface inversion pattern

When A depends on B's behavior but not its implementation:

- Define an abstract interface in a lower layer.
- Let B implement it.
- A depends only on the abstraction.

4.1.4 Real-world module architecture examples

Example: Service-oriented backend

`acme.types`
`acme.core.logic`
`acme.core.validation`
`acme.persistence`
`acme.network`
`acme.api`
`acme.app`

Dependencies:

- `acme.types` imported by all.
- `acme.core.logic` imports types.
- `acme.persistence` imports types.
- `acme.api` imports core + types.
- `acme.app` imports api + network.

No cycles exist. Each module has a clear role.

Example: Game engine structure

```
engine.math
engine.core
engine.render
engine.audio
engine.physics
engine.runtime
game.logic
```

- Foundational math types are isolated.
- Rendering does not import physics.
- Runtime coordinates subsystems.

4.2 Refactoring header-based projects to modules

4.2.1 Identifying import boundaries

Migration begins by identifying:

- Stable headers used widely.
- Natural ownership clusters (domain features).
- Headers that define public contracts vs internal helpers.

Step 1: Draw include graph

Generate a dependency graph of headers. Identify:

- Highly included headers (good module candidates).
- Deep include chains.
- Cycles.

Step 2: Define primary module boundaries

Create module interfaces corresponding to:

- Major subsystems.
- Public library APIs.
- Domain features.

4.2.2 Handling macros safely

Macros are the most dangerous part of migration.

Rules

- Eliminate configuration macros from public headers.
- Move macro logic into implementation units or build system.
- If unavoidable, isolate in a global module fragment.

Example: isolating macro-based configuration

```
module;

#define ACME_INTERNAL_FEATURE 1
#include "legacy_config.hpp"
```

```
export module acme.config;

export namespace acme {
  export constexpr bool feature_enabled = true;
}
```

The macro does not leak into importers.

4.2.3 Refactoring STL-heavy code

STL-heavy headers often cause build bloat.

Technique 1: Move containers to implementation units

Instead of exposing:

```
export std::vector<int> compute();
```

Expose:

```
export struct Result {
  int count;
};
```

Or:

```
export void compute_into(std::span<int>);
```

Technique 2: Partition heavy algorithms

Move template-heavy algorithms into internal partitions. Export only stable entry points.

4.2.4 Migrating build systems

Build systems must:

- Discover module dependencies.
- Compile module interfaces before importers.
- Track compiler flags consistently.

General migration plan

1. Upgrade to a module-aware build system version.
2. Introduce one module at a time.
3. Keep mixed header/module mode during transition.
4. Enforce no cycles in module graph.

4.3 Module ABI & binary interface considerations

4.3.1 How modules impact ABI

Modules affect:

- Source compilation boundaries.
- Visibility of declarations.
- Dependency structure.

They do **not** define:

- Object layout rules.
- Calling conventions.
- Name mangling formats.

Therefore:

> Modules improve encapsulation but do not automatically guarantee ABI stability.

4.3.2 Versioning modules

Two approaches:

1) Namespace versioning

```
export module acme.api;

export namespace acme::v2 {
  export int run();
}
```

2) Module renaming

`acme.api.v1`

`acme.api.v2`

Namespace versioning keeps module name stable. Module renaming isolates compilation artifacts.

4.3.3 Compatibility guarantees

To maintain compatibility:

- Do not remove exported symbols without versioning.
- Preserve layout of exported structs if ABI stability is required.
- Avoid changing virtual table shape across releases.
- Avoid adding exported inline functions that alter behavior unexpectedly.

4.3.4 Module boundaries in DLL/shared libraries

When combining modules with shared libraries:

- Ensure exported symbols use correct visibility attributes.
- Avoid exposing inline-heavy templates across DLL boundaries.
- Keep module boundary aligned with binary boundary when possible.
- Test cross-version loading scenarios.

Example: visibility control

```
export module acme.shared;

#if defined(_WIN32)
  #define ACME_API __declspec(dllexport)
#else
  #define ACME_API
#endif

export namespace acme {
  export class ACME_API Service {
  public:
    void run();
```

```
  };
}
```

Even in module-based designs, binary visibility remains a platform concern.

4.4 Engineering checklist for large projects

1. Separate API from implementation early.
2. Keep module interfaces small and stable.
3. Avoid import cycles.
4. Align module graph with architectural layering.
5. Treat ABI boundaries independently from module boundaries.
6. Centralize macro and platform logic.
7. Validate module dependency DAG regularly.

4.5 Exercises

Exercise 1

Refactor a header-only library into:

- one primary interface,
- one exported partition,
- one private partition.

Exercise 2

Draw a module dependency DAG for your current project. Eliminate at least one cycle.

Exercise 3

Design a module that is safe to expose from a shared library and verify ABI stability under a minor internal change.

Part III

Build Systems, Toolchains & Multi-Platform Engineering

The C++ Build Pipeline (Deeper Than Volume 1)

5.1 Compiler front-ends and module compilation

5.1.1 What changes in the pipeline when you enable modules

In the classic header model, the compiler front-end repeatedly performs the same expensive work for every translation unit (TU):

1. run the preprocessor (`#include`, macros),
2. parse and build the AST,
3. perform semantic analysis and template instantiation,
4. generate an object file,
5. repeat for the next TU.

With C++20 named modules, the pipeline gains a new compilation artifact:

- a **Binary Module Interface** (BMI), generated when compiling a module interface unit,
- later **consumed by importers** during compilation (front-end time), not during linking.

A practical mental model:

- **Interface compilation** produces *(object file + BMI)*.
- **Importer compilation** consumes *BMI* to perform semantic import of exported declarations and produces its own object file.
- **Linking** links object files as usual; BMIs are *not* link inputs.

The build-order constraint

Header inclusion needs no global compilation order. Modules require ordering:

- module interfaces must be compiled before any TU that imports them,
- module partitions have their own order constraints,
- a build system must either know or discover the module dependency graph.

The "chicken-and-egg" problem

A build tool must know module dependencies to schedule compilation, but dependencies are discovered by parsing/compiling sources. Modern module-aware build systems solve this by:

- scanning sources for `import` and module declarations,
- producing dependency metadata,
- dynamically updating the build graph with the correct build order.

5.1.2 A shared vocabulary: interface units, implementation units, partitions, BMI

We will use the following terminology consistently:

- **Primary module interface unit**: contains `export module M;` and exports the main API.
- **Module interface partition**: a partition unit that contributes interface components, often exported.
- **Module implementation unit**: contains `module M;` or `module M:part;` and provides definitions.
- **BMI**: compiled representation of a module interface (format is compiler-specific).

5.1.3 MSVC module model

MSVC implements C++ modules by emitting:

- an object file (`.obj`) for code generation,
- an **IFC** file (`.ifc`) that acts as the persistent module interface representation (BMI concept).

How the MSVC front-end uses IFC

Conceptually, the front-end:

- parses a module interface unit,
- builds semantic graphs/AST-like internal representation,
- serializes exportable semantic information to `.ifc`,
- later importers memory-map or load IFC to obtain exported declarations efficiently.

A minimal MSVC-flavored workflow (conceptual command sequence)

```
:: Compile module interface (produces .ifc + .obj)
cl /std:c++20 /EHsc /c /interface acme.math.cppm

:: Compile importer (consumes BMI discovered by build system or via flags)
cl /std:c++20 /EHsc /c main.cpp

:: Link as usual (BMI is not a link input)
link main.obj acme.math.obj /OUT:app.exe
```

Engineering notes:

- MSVC requires consistent compilation settings for producers and consumers (standard mode, important macros, include paths).
- The BMI file is **not** a stable cross-toolchain artifact; it is toolchain- and often version-dependent.
- Treat IFC/BMI as a build-cache artifact: generated and consumed within a controlled build environment.

MSVC-specific stability principle

If you want stable builds, ensure:

- all files that import a given module are compiled with compatible flags,
- the build system uses a single coordinated module cache directory per configuration,
- the same compiler version is used for both BMI production and BMI consumption.

5.1.4 Clang module model

Clang's standard C++ modules support typically uses:

- **PCM** files (`.pcm`) as BMI-like artifacts for compiled module interfaces,
- object files (`.o`) for code generation.

How Clang consumes module BMIs

Common usage patterns include:

- emitting BMIs during interface compilation,
- using a "prebuilt module path" to find BMIs when compiling importers,
- mapping module names to BMI files when necessary.

A minimal Clang-flavored workflow (conceptual)

```
# Compile module interface to produce BMI (.pcm) and object file
clang++ -std=c++20 -c acme.math.cppm -o acme.math.o

# Compile importer with BMI discoverability (often via build-system-provided flags)
clang++ -std=c++20 -c main.cpp -o main.o

# Link as usual
clang++ main.o acme.math.o -o app
```

Engineering notes:

- The exact flags vary by Clang version and build system strategy.
- Clang often requires explicit paths/mappings to locate the correct `.pcm` for a module name.
- Keep the module cache isolated per build configuration to avoid stale or mismatched BMIs.

The “PCM locking” class of problems (tooling interoperability)

In real-world environments, background tools (indexers, language servers) may open module artifacts while the compiler attempts to overwrite them. Engineering mitigations:

- separate build and index caches (different output directories),
- avoid sharing the same BMI location across concurrent build tools,
- ensure your build system owns the BMI generation paths deterministically.

5.1.5 GCC module model

GCC implements C++ modules with its own BMI artifacts. In practice you will encounter:

- **GCM** files (commonly `.gcm`) as GCC’s compiled module interface artifact,
- object files (`.o`) for generated code.

GCC front-end realities

GCC modules have evolved across versions, and build integration has historically required careful coordination. Key engineering rules remain stable:

- module interface BMIs must be built before importers,
- consumers must be compiled with consistent options,
- BMI locations must be discoverable in a deterministic way.

A minimal GCC-flavored workflow (conceptual)

```
# Compile module interface to generate BMI and object
g++ -std=c++20 -c acme.math.cppm -o acme.math.o

# Compile importer
g++ -std=c++20 -c main.cpp -o main.o

# Link
g++ main.o acme.math.o -o app
```

Engineering notes:

- GCC's module support has had different flags and behaviors across releases; rely on a module-aware build system rather than hand-assembling flags.
- As with other compilers, do not treat `.gcm` as a distributable ABI artifact.

5.1.6 BMI formats across compilers

BMIs are a shared *concept*, not a shared format. The actual BMI artifact differs:

- MSVC: **IFC** (`.ifc`) as the persisted semantic interface representation.
- Clang: **PCM** (`.pcm`) as the compiled module interface artifact.
- GCC: commonly **GCM** (`.gcm`) as the compiled module interface artifact.

Engineering consequences

1. **BMI incompatibility is expected**: you cannot build BMIs with one compiler and import them with another.

2. **BMI versioning is real**: a BMI produced by one version of a compiler may not be consumable by another version.

3. **Packaging must treat BMIs as build outputs**: distribute sources (or distribute prebuilt binaries with a controlled toolchain), but do not assume BMIs are portable deliverables.

A robust portability policy

For multi-platform and multi-toolchain projects:

- keep modules as a **source organization mechanism**,
- treat BMI emission as an **internal build detail**,
- package libraries as source (for consumers who build) or as binaries with clear ABI/toolchain constraints.

5.1.7 Front-end scanning and the build system

Large projects cannot reliably compile modules without build-system cooperation. A module-aware build system typically:

- scans sources to discover module declarations and imports,
- produces a dependency graph,
- schedules interface compilation before importers,
- assigns BMI output paths and provides BMI lookup flags to the compiler.

Practical example: a module-aware target design

Even if the syntax differs across build tools, the architectural structure is consistent:

- **Producers**: module interface units and partitions.
- **Consumers**: implementation units and ordinary TUs that import modules.
- **Artifact routing**: BMI outputs to a per-config cache directory, objects to normal object directories.

Example project structure for scanning

```
src/
  libs/math/
    acme.math.cppm
    acme.math:types.cppm
    acme.math.impl.cpp
  app/
    main.cpp
```

5.2 Linking with modules

5.2.1 A critical truth: modules are a front-end feature

To avoid architectural confusion:

> `import` is resolved by the compiler front-end at compile time. The linker does not "import modules".

What the linker sees is the same as before:

- object files (`.o`/`.obj`),
- static libraries (`.a`/`.lib`),
- shared libraries (`.so`/`.dylib`/`.dll` + import lib),
- optional LTO bitcode payloads inside objects (depending on toolchain).

Therefore:

- module syntax shapes *what gets compiled* and *how fast and reliably*,
- linking remains about symbol resolution, relocation, visibility, and (optionally) whole-program optimization.

5.2.2 Symbol visibility

Symbol visibility is still controlled by:

- language linkage (mangling),
- compile-time visibility attributes (e.g., default/hidden visibility on ELF platforms),
- export/import declarations on Windows (DLL boundary),
- the build system's choice of static vs shared library boundaries.

Modules do not replace visibility control

Exporting a name from a module makes it reachable by importers at compile time, but that is not the same as exporting a symbol from a shared library. You can have:

- an exported module API that is linked statically into an executable (no shared export needed),

- a shared library that exports only a subset of its compiled module code via visibility rules,
- a module that spans multiple libraries (possible, but usually painful and not recommended).

Practical rule: align module boundaries with link boundaries when possible

For large systems:

- keep a module's implementation and interface within a single library target when feasible,
- use one facade module per library as a stable import point,
- avoid splitting one named module across multiple libraries unless you fully control the consumer build graph.

Visibility example (cross-platform pattern)

```
// File: acme.shared.api.cppm
export module acme.shared.api;

#if defined(_WIN32)
  #if defined(ACME_SHARED_BUILD)
    #define ACME_API __declspec(dllexport)
  #else
    #define ACME_API __declspec(dllimport)
  #endif
#else
  #define ACME_API __attribute__((visibility("default")))
#endif

export namespace acme {
  export class ACME_API Service {
  public:
```

```
    void run();
  };
}
```

This remains necessary because binary export is a platform ABI concern, not a module concern.

5.2.3 Linker variations (`ld` / `ld.gold` / `lld` / MSVC `link.exe`)

Different linkers implement the same core responsibilities but differ in:

- link-time performance and memory behavior,
- diagnostic quality,
- default behaviors for dead-strip, ICF, and incremental linking,
- LTO integration quality and supported modes.

ELF world: `ld.bfd`, `gold`, `lld`

On Linux-like platforms:

- `ld.bfd` is the traditional GNU linker, widely compatible.
- `gold` was designed for speed but is less central today.
- `lld` (LLVM linker) is often fastest and integrates well with LLVM LTO pipelines.

Windows world: MSVC `link.exe`

On Windows:

- `link.exe` supports incremental linking and PDB generation workflows,
- symbol export is strongly tied to DLL boundaries and import libraries,
- whole-program optimization integrates with the MSVC toolchain.

What modules change for linkers

Modules typically do *not* change the linker's job. They can indirectly change the link step by:

- shifting code from headers (inline/templates) into implementation units (changing which objects contain which symbols),
- reducing redundant instantiations (potentially reducing object size),
- enabling cleaner library boundaries (which improves incremental link behavior in practice).

5.2.4 Link-time optimization (LTO)

LTO is about optimizing across object boundaries by feeding intermediate representation (IR) through the optimizer at link time. Key variants:

- **Full LTO**: whole program (or whole link unit) is optimized together.
- **ThinLTO**: scalable, incremental-friendly variant (common in LLVM pipelines).

How modules interact with LTO

Modules primarily improve front-end scalability (parsing/semantic reuse), while LTO improves back-end optimization across translation units. They are complementary:

- modules can reduce the need for massive header inlining to achieve performance,
- you can keep interfaces clean and move code into implementation units,
- LTO can still inline across those units when beneficial.

Engineering strategy: “move out of headers, rely on LTO for inlining”

A practical high-performance strategy in large projects:

1. export small stable APIs from modules,
2. implement heavy logic in implementation units,
3. enable LTO/ThinLTO in release builds,
4. keep debug builds fast by avoiding excessive template/header bloat.

Example: reducing header/template exposure while keeping performance

```
// acme.fastmath.cppm
export module acme.fastmath;

export namespace acme::fastmath {
  export double dot(const double* a, const double* b, int n) noexcept;
}
```

```
// acme.fastmath.impl.cpp
module acme.fastmath;

namespace acme::fastmath {
  double dot(const double* a, const double* b, int n) noexcept {
    double s = 0.0;
    for (int i = 0; i < n; ++i) s += a[i] * b[i];
    return s;
  }
}
```

With LTO enabled, the compiler may inline or vectorize across call boundaries, but you avoid the rebuild cost and dependency fan-out of header-only exposure.

5.2.5 Module importing at link-time

This phrase appears frequently in casual discussions, but it is technically misleading. Clarify the correct model:

What is *not* happening

- the linker does not parse `import` statements,
- the linker does not read BMIs to resolve names,
- link order does not determine module visibility.

What *is* happening

- compile-time import determines which declarations are available during compilation,
- those declarations drive emitted symbols and references in object files,
- the linker resolves those references in the usual symbol-resolution model.

A concrete example: import affects compilation, not linking

```
// acme.math.cppm
export module acme.math;
export int add(int, int);
```

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

```
// main.cpp
import acme.math;
int main() { return add(1,2); }
```

Pipeline:

1. compile `acme.math.cppm` → BMI + object (producer)
2. compile `main.cpp` with BMI available → object (consumer)
3. link objects → executable

The linker never learns about modules; it only sees that `main.o` references `add` and that `acme.math.o` defines it.

5.3 Practical build engineering guidance for large module projects

5.3.1 A disciplined directory and target layout

A scalable layout that works across platforms:

```
src/
  libs/
    core/
      module/          (primary interfaces and partitions)
      impl/            (implementation units)
      include/         (optional public headers for non-module consumers)
    infra/
      platform/
      net/
  app/
    cli/
    server/
tests/
```

5.3.2 Hard rules that prevent 80% of real-world pain

1. Keep module interface units small and stable.
2. Put heavy dependencies in implementation units or private partitions.
3. Centralize platform macros in a few boundary modules (platform facade).
4. Never share BMI cache directories across different compiler versions or configurations.
5. Let the build system manage BMI output paths and BMI lookup flags.

5.3.3 A module build checklist

- Do module producers and consumers use the same `-std=` mode?
- Are the same critical feature macros applied consistently?
- Is BMI output per build configuration isolated?
- Are there any import cycles (direct or indirect)?
- Are you mixing "include-order contracts" with module imports?

5.4 Exercises

Exercise 1: Observe producer/consumer ordering

1. Create two modules `A` and `B` where `A` imports `B`.
2. Attempt to compile `A` before `B` and observe the failure.
3. Configure your build so the dependency order is discovered automatically.

Exercise 2: Separate BMI caches per configuration

1. Build Debug and Release configurations with modules enabled.
2. Verify that BMIs do not overwrite each other.
3. Intentionally reuse the same BMI directory and observe the instability it can cause.

Exercise 3: Linkers and LTO with module-based code

1. Create a module interface that exports a small API with heavy implementation in an implementation unit.
2. Build without LTO and measure runtime performance.
3. Enable LTO/ThinLTO (depending on toolchain) and measure whether cross-unit inlining occurs.

Advanced CMake for Real-World Projects

6.1 Target-Based Design Principles

6.1.1 Why modern CMake is target-first

Modern CMake is not a "directory scripting language"; it is a **target graph generator**. The primary unit of design is a **target** (an executable, static library, shared library, object library, interface-only library). A professional project treats targets as first-class architectural objects:

- **Targets own their build rules**: include paths, compile options, link options, preprocessor definitions, language standards.
- **Targets express dependencies explicitly**: you link to other targets, not to raw flags.
- **Targets are composable**: higher-level targets consume lower-level targets through transitive usage requirements.

6.1.2 Target ownership

Target ownership means:

1. A target is the **only** place where its compilation model is defined.
2. A target should be buildable and testable as a unit.

3. A target should expose a deliberate public surface to dependents.

Architectural target layout

A scalable structure typically maps to layers:

```
acme::core        (pure primitives, no OS dependencies)
acme::domain      (business rules, minimal dependencies)
acme::infra       (OS/db/net adapters)
acme::app         (executables / entrypoints)
```

Each library target is a boundary for:

- compile definitions,
- include visibility,
- link visibility,
- module export/import policy (C++20 modules).

6.1.3 Transitive usage requirements

In target-based design, dependencies propagate through **usage requirements**. If a library needs an include directory or a compile definition for *its users* to compile correctly, that requirement must be expressed transitively.

Usage requirements categories

- **Compile interface**: includes, compile definitions, compile options required by consumers.
- **Link interface**: libraries and link options required by consumers.

Correct mental model

When a consumer does:

`target_link_libraries(consumer PRIVATE producer)`

the consumer receives the producer's **usage requirements** according to the link visibility keyword.

6.1.4 PUBLIC / PRIVATE / INTERFACE

These keywords are not cosmetic. They define the contract boundary.

Meaning

- **PRIVATE**: used to build the target itself; not required by consumers.
- **PUBLIC**: used to build the target and also required by consumers.
- **INTERFACE**: not used to build the target (because it has no sources or the property is purely usage); required by consumers.

Example: include visibility

```
add_library(acme_core STATIC)
target_sources(acme_core PRIVATE
  src/core/id.cpp
  src/core/error.cpp
)

target_include_directories(acme_core
  PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
```

```
  PRIVATE
    ${CMAKE_CURRENT_SOURCE_DIR}/src
)

target_compile_features(acme_core PUBLIC cxx_std_20)
```

Interpretation:

- Public headers live in `include/` and are required by dependents.
- Private includes for implementation details live in `src/` and do not propagate.
- The target requires C++20 for consumers (because public headers or exported modules require it).

Example: compile definitions contract

```
target_compile_definitions(acme_core
  PUBLIC  ACME_CORE_API_LEVEL=2
  PRIVATE ACME_CORE_INTERNAL_BUILD=1
)
```

Rule:

- If a public header or module interface depends on a macro, it must be **PUBLIC**.
- If it is only for internal compilation, it must be **PRIVATE**.

Example: INTERFACE library for policy

A pure policy target can coordinate flags or toolchain requirements:

```
add_library(acme_warnings INTERFACE)
```

```
target_compile_options(acme_warnings INTERFACE
  $<$<CXX_COMPILER_ID:MSVC>:/W4 /permissive->
  $<$<NOT:$<CXX_COMPILER_ID:MSVC>>:-Wall -Wextra -Wpedantic>
)

target_link_libraries(acme_core PUBLIC acme_warnings)
```

This avoids copy-pasting flags across many targets and keeps policy centralized.

6.2 Dependency Management

6.2.1 A dependency strategy that scales

Professional projects choose a dependency strategy explicitly, usually combining:

- **System packages** via `find_package` (preferred for stable, system-provided libraries).
- **Vendored sources** via `FetchContent` (preferred for reproducible builds in CI).
- **Package managers** (Conan, vcpkg) to standardize acquisition and binary caching.
- **Exported CMake packages** for your own libraries (so others can consume you correctly).

6.2.2 FetchContent deep dive

The real value: reproducibility with target-based integration

`FetchContent` is best when:

- you want exact source versions checked by hashes/tags,
- you want CI reproducibility,
- you want to build dependencies with your toolchain and flags.

The correct pattern: declare once, make available, link to targets

```
include(FetchContent)

FetchContent_Declare(
  fmt
  GIT_REPOSITORY https://github.com/fmtlib/fmt.git
  GIT_TAG        10.2.1
)

FetchContent_MakeAvailable(fmt)

target_link_libraries(acme_core PUBLIC fmt::fmt)
```

Professional notes:

- Link to exported targets (e.g., `fmt::fmt`) not to raw include paths.
- Prefer `FetchContent_MakeAvailable` so dependency targets are defined correctly.

Controlling options of a fetched project

Many dependencies expose options. Set them **before** `FetchContent_MakeAvailable`:

```
set(FMT_DOC OFF CACHE BOOL "" FORCE)
set(FMT_TEST OFF CACHE BOOL "" FORCE)

FetchContent_MakeAvailable(fmt)
```

Avoiding duplicate dependency instances

The largest real-world problem with `FetchContent` is accidental duplication:

- your project fetches a dependency,

- a subdependency fetches the same dependency differently,
- you end up with two copies and ODR/ABI confusion.

Mitigation strategy:

- centralize `FetchContent` in a top-level `cmake/deps.cmake`,
- do not let leaf targets fetch dependencies,
- expose dependency targets through your own umbrella targets (e.g., `acme_deps`).

FetchContent + find_package redirection (professional pattern)

A powerful pattern is: allow `find_package` for system installs, but fall back to `FetchContent`.

```
find_package(fmt CONFIG QUIET)

if(NOT fmt_FOUND)
  include(FetchContent)
  FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG        10.2.1
  )
  FetchContent_MakeAvailable(fmt)
endif()

target_link_libraries(acme_core PUBLIC fmt::fmt)
```

This enables:

- fast local builds when the package is installed,
- reproducible CI builds when it is not.

6.2.3 find_package in detail

Two modes: Config vs Module

`find_package` can find packages in two major ways:

- **Config mode**: finds `<Package>Config.cmake` provided by the package itself (best practice).
- **Module mode**: finds `Find<Package>.cmake` provided by CMake or your project (fallback/legacy).

Preferred: Config mode with imported targets

In modern ecosystems, a good package provides imported targets:

```
find_package(ZLIB REQUIRED)
target_link_libraries(acme_infra PRIVATE ZLIB::ZLIB)
```

Versioning and components

```
find_package(Boost 1.84 REQUIRED COMPONENTS filesystem system)

target_link_libraries(acme_infra
  PRIVATE
    Boost::filesystem
    Boost::system
)
```

Professional rule:

- Prefer imported targets over variables like `Boost_LIBRARIES`.
- Use `REQUIRED` for hard dependencies to fail early.

Toolchain and search path discipline

In real projects, dependency resolution becomes chaotic unless you decide:

- whether you prefer system packages or vendored packages,
- whether you allow user overrides via `CMAKE_PREFIX_PATH`,
- how CI chooses consistent versions.

A good policy is:

- CI uses a pinned toolchain and pinned dependency set,
- local developers may use system packages for convenience,
- any deviation must be explicit (via presets or toolchain files).

6.2.4 Conan & vcpkg integration

Conan integration model (practical)

In modern workflows, Conan typically integrates by generating:

- a toolchain file (to set compilers, flags, runtime, sysroots),
- dependency config files (so `find_package` works),
- imported targets for libraries.

CMake-side policy:

- treat the Conan toolchain file as part of the configuration preset,
- keep `find_package` usage normal and target-based,
- never manually propagate include directories or library paths produced by Conan.

vcpkg integration model (practical)

vcpkg commonly integrates via a toolchain file that:

- sets up `CMAKE_PREFIX_PATH` to find installed packages,
- enables `find_package(<Pkg> CONFIG)` to succeed,
- provides consistent triplets (x64-windows, x64-linux, arm64-osx, etc.).

CMake-side policy:

- provide a preset that points to the vcpkg toolchain file,
- keep consumption target-based (imported targets),
- avoid mixing vcpkg and FetchContent for the same dependency unless you have strict isolation.

6.2.5 Exporting your own CMake packages

Goal: consumers do this

```
find_package(acme_core CONFIG REQUIRED)
target_link_libraries(their_app PRIVATE acme::core)
```

And everything they need propagates automatically:

- include directories,
- compile features,
- required dependencies (as imported targets),
- platform-specific link options when needed.

Step-by-step professional export pattern

Assume you have a target `acme_core` and you want to export it as `acme::core`.

```
add_library(acme_core STATIC)
add_library(acme::core ALIAS acme_core)

target_sources(acme_core PRIVATE
  src/id.cpp
  src/error.cpp
)

target_include_directories(acme_core
  PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
)

target_compile_features(acme_core PUBLIC cxx_std_20)
```

Install the target and export set:

```
include(GNUInstallDirs)

install(TARGETS acme_core
  EXPORT acmeTargets
  ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
  LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
  RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
  INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)

install(DIRECTORY include/
  DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
```

```
)

install(EXPORT acmeTargets
  FILE acmeTargets.cmake
  NAMESPACE acme::
  DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/acme
)
```

Generate and install the package config and version files:

```
include(CMakePackageConfigHelpers)

configure_package_config_file(
  ${CMAKE_CURRENT_SOURCE_DIR}/cmake/acmeConfig.cmake.in
  ${CMAKE_CURRENT_BINARY_DIR}/acmeConfig.cmake
  INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/acme
)

write_basic_package_version_file(
  ${CMAKE_CURRENT_BINARY_DIR}/acmeConfigVersion.cmake
  VERSION 2.0.0
  COMPATIBILITY SameMajorVersion
)

install(FILES
  ${CMAKE_CURRENT_BINARY_DIR}/acmeConfig.cmake
  ${CMAKE_CURRENT_BINARY_DIR}/acmeConfigVersion.cmake
  DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/acme
)
```

Minimal `acmeConfig.cmake.in`:

```
@PACKAGE_INIT@

include("${CMAKE_CURRENT_LIST_DIR}/acmeTargets.cmake")
```

Professional notes:

- Your exported package must reference exported targets, not raw variables.
- If your library depends on third-party targets, your `acmeConfig.cmake` may also need `find_dependency` logic in a controlled way.

6.3 CMake + Modules

6.3.1 Scanning module dependencies

Named modules require build ordering. CMake supports module dependency discovery by:

- asking the compiler to scan sources for module dependencies,
- collecting scanning results to infer build-order constraints,
- telling the build tool how to dynamically update the build graph.

This implies:

- the generator must support dynamic build graph updates for modules,
- module-capable compilers are required,
- the build must be structured so module interface sources are declared correctly.

FILE_SET of type CXX_MODULES (the key requirement)

In CMake's module model, any module-providing source file must be placed in a **file set** of type `CXX_MODULES`. This is the canonical pattern:

```
add_library(acme_math STATIC)

target_sources(acme_math
  PUBLIC
    FILE_SET cxx_modules TYPE CXX_MODULES FILES
      src/acme.math.cppm
      src/acme.math:types.cppm
  PRIVATE
      src/acme.math.impl.cpp
)

target_compile_features(acme_math PUBLIC cxx_std_20)
```

Interpretation:

- The module interface units and exported partitions live in the CXX_MODULES file set.
- Implementation units remain regular sources (PRIVATE).
- Consumers link to `acme_math` normally and import the module in C++ code.

6.3.2 Handling BMI files

BMIs are compiler-specific compiled interface artifacts. CMake treats them as build outputs, typically managed automatically by the generator and compiler integration.

Engineering realities of BMIs

- BMIs are not portable across compilers and often not across compiler versions.
- BMIs must be isolated by build configuration (Debug/Release) and toolchain.
- You must avoid sharing the same BMI cache directory between different configurations or different compiler versions.

Practical directory discipline

Use out-of-source builds and isolate by preset:

```
out/
  build/msvc-debug/
  build/msvc-release/
  build/clang-debug/
  build/gcc-release/
```

This automatically prevents BMI collisions and cache poisoning.

A module-safe target boundary rule

> Keep the producers and consumers of module BMIs within a single coherent build configuration.

Do not attempt to:

- build BMIs once and reuse them across multiple independent build directories,
- ship BMIs as part of a binary package for arbitrary consumer toolchains.

6.3.3 Multi-compiler compatibility

Multi-compiler module builds are achievable, but only if you accept that:

- module build mechanics differ across MSVC/Clang/GCC,
- the compiler version matters more than in the header-only world,
- the build system must own the scanning and ordering.

A practical compatibility strategy

1. **Choose a baseline**: decide the minimum compiler versions for MSVC, Clang, GCC.
2. **Use a module-capable generator**: do not depend on legacy generators for modules.
3. **Centralize policies** in CMake:
    - standard level (`cxx_std_20` or newer),
    - warning policy,
    - feature toggles,
    - standard library import strategy (if used).
4. **Avoid compiler-specific module flags in user code**.
5. **Test all toolchains in CI** with the same module target graph.

6.3.4 Cross-platform module builds

A cross-platform module build must also account for:

- platform macros and platform headers (`windows.h`, POSIX headers),
- file system case sensitivity affecting module naming conventions,
- path length limits on Windows in deeply nested build trees,
- differences in standard library module availability across toolchains.

Platform boundary modules

A reliable pattern is to isolate platform differences behind facade modules:

```
// src/acme.platform.cppm
export module acme.platform;

#if defined(_WIN32)
export import acme.platform.win;
#else
export import acme.platform.posix;
#endif

export namespace acme::platform {
  export void init();
}
```

Keep such conditional imports centralized (in a few boundary modules), not scattered throughout core modules.

6.4 Professional CMake Presets

6.4.1 Why presets are a professional requirement

Presets transform configuration from “tribal knowledge” into an explicit, version-controlled workflow.
They allow:

- repeatable configure/build/test invocations,
- consistent compiler/toolchain selection,
- consistent dependency policy,

- clean CI integration.

6.4.2 Configure workflows

A configure preset defines:

- generator,
- binary directory,
- cache variables (toolchain file, options),
- environment variables,
- toolchain constraints.

Professional `CMakePresets.json` skeleton

```
{
  "version": 6,
  "configurePresets": [
    {
      "name": "base",
      "hidden": true,
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/out/build/${presetName}",
      "cacheVariables": {
        "CMAKE_EXPORT_COMPILE_COMMANDS": "ON",
        "CMAKE_CXX_STANDARD": "20",
        "CMAKE_CXX_STANDARD_REQUIRED": "ON",
        "CMAKE_CXX_EXTENSIONS": "OFF"
      }
    },
    {
```

```
    "name": "clang-debug",
    "inherits": "base",
    "cacheVariables": {
      "CMAKE_BUILD_TYPE": "Debug",
      "CMAKE_CXX_COMPILER": "clang++"
    }
  },
  {
    "name": "gcc-release",
    "inherits": "base",
    "cacheVariables": {
      "CMAKE_BUILD_TYPE": "Release",
      "CMAKE_CXX_COMPILER": "g++"
    }
  }
],
"buildPresets": [
  {
    "name": "clang-debug",
    "configurePreset": "clang-debug"
  },
  {
    "name": "gcc-release",
    "configurePreset": "gcc-release"
  }
],
"testPresets": [
  {
    "name": "clang-debug",
    "configurePreset": "clang-debug",
    "output": {
      "outputOnFailure": true
    }
```

```
    }
  ]
}
```

Professional notes:

- Keep `base` hidden and inherit from it.
- Make the build directory include the preset name to isolate caches (critical for modules).
- Encode policy once (standard, extensions, compile commands).

6.4.3 Build presets

Build presets define:

- which configure preset they correspond to,
- configuration selection for multi-config generators,
- build targets, parallelism, and verbosity options.

Common professional additions

```
{
  "name": "clang-debug-fast",
  "configurePreset": "clang-debug",
  "jobs": 16,
  "targets": [ "all" ]
}
```

6.4.4 Test presets

Test presets define:

- which configure preset they correspond to,
- CTest output behavior,
- filtering, parallelism, repeat strategies.

Example: stable CI test behavior

```
{
  "name": "ci-tests",
  "configurePreset": "gcc-release",
  "execution": {
    "noTestsAction": "error",
    "stopOnFailure": false
  },
  "output": {
    "outputOnFailure": true
  }
}
```

6.4.5 Preset portability

Presets are portable when:

- they avoid absolute paths,
- toolchain selection is expressed as variables or environment,
- external toolchain files are referenced relative to the source directory,
- OS-specific differences are isolated in OS-specific presets.

Portable toolchain file reference

```
{
  "name": "vcpkg-msvc",
  "inherits": "base",
  "cacheVariables": {
    "CMAKE_TOOLCHAIN_FILE": "${sourceDir}/external/vcpkg/scripts/buildsystems/vcpkg.cmake"
  }
}
```

Professional pattern:

- Keep `CMakePresets.json` in source control for shared workflows.
- Use `CMakeUserPresets.json` for developer-local overrides (paths, personal toolchains).

6.5 A complete reference template for a real module-enabled library

This example combines:

- target-based design,
- exported package,
- module file sets,
- dependency consumption via `find_package`.

Top-level `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.28)
```

```
project(acme
  VERSION 2.0.0
  LANGUAGES CXX
)

include(GNUInstallDirs)
include(CMakePackageConfigHelpers)

add_library(acme_core STATIC)
add_library(acme::core ALIAS acme_core)

target_compile_features(acme_core PUBLIC cxx_std_20)

target_sources(acme_core
  PUBLIC
    FILE_SET cxx_modules TYPE CXX_MODULES FILES
      src/acme.core.cppm
      src/acme.core:types.cppm
  PRIVATE
      src/acme.core.impl.cpp
)

target_include_directories(acme_core
  PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
)

find_package(fmt CONFIG QUIET)
if(NOT fmt_FOUND)
  include(FetchContent)
  FetchContent_Declare(
    fmt
```

```
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 10.2.1
  )
  FetchContent_MakeAvailable(fmt)
endif()

target_link_libraries(acme_core PUBLIC fmt::fmt)

install(TARGETS acme_core
  EXPORT acmeTargets
  ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
  LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
  RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
  INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)

install(DIRECTORY include/
  DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)

install(EXPORT acmeTargets
  FILE acmeTargets.cmake
  NAMESPACE acme::
  DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/acme
)

configure_package_config_file(
  ${CMAKE_CURRENT_SOURCE_DIR}/cmake/acmeConfig.cmake.in
  ${CMAKE_CURRENT_BINARY_DIR}/acmeConfig.cmake
  INSTALL_DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/acme
)

write_basic_package_version_file(
```

```
  ${CMAKE_CURRENT_BINARY_DIR}/acmeConfigVersion.cmake
  VERSION ${PROJECT_VERSION}
  COMPATIBILITY SameMajorVersion
)

install(FILES
  ${CMAKE_CURRENT_BINARY_DIR}/acmeConfig.cmake
  ${CMAKE_CURRENT_BINARY_DIR}/acmeConfigVersion.cmake
  DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/acme
)
```

cmake/acmeConfig.cmake.in

```
@PACKAGE_INIT@

include("${CMAKE_CURRENT_LIST_DIR}/acmeTargets.cmake")
```

6.6 Professional checklist (Chapter 6)

1. Every library is a target. No global include directories.
2. No global compile definitions. All requirements belong to targets.
3. Every dependency is consumed via imported targets.
4. `PUBLIC/PRIVATE/INTERFACE` is used deliberately, not randomly.
5. Modules: module-providing sources are declared in `FILE_SET CXX_MODULES`.
6. Module build directories are isolated by preset to prevent BMI cache collisions.
7. Presets exist for:

- at least one Debug config,
- at least one Release config,
- at least one toolchain per platform/compiler family,
- at least one CI test preset.

8. Your own libraries export a relocatable CMake package.

6.7 Exercises

Exercise 1: Fix a broken transitive dependency

Create two libraries `A` and `B` where `B` depends on `A`. Make `B` compile but `B`'s consumer fail, then fix the issue by moving a requirement from `PRIVATE` to `PUBLIC`.

Exercise 2: Create a portable preset matrix

Create presets for:

- GCC Debug/Release,
- Clang Debug/Release,
- one Windows MSVC preset,
- one preset using a toolchain file.

Verify that each preset builds into a unique directory under `out/build`.

Exercise 3: Convert one header library into a module target

Pick a small header-based component and:

1. create a `.cppm` interface unit exporting a minimal API,
2. move implementation to `.cpp` implementation units,
3. declare the interface in a `CXX_MODULES` file set,
4. ensure consumers use `import` instead of `#include`.

Tooling for Large-Scale Codebases (Advanced & Expanded)

7.1 clang-tidy Advanced Use (Deep Engineering Perspective)

7.1.1 clang-tidy as an AST-Driven Refactoring Engine

`clang-tidy` operates on Clang's semantic model, not on raw text. This distinction is fundamental in large-scale C++ systems because:

- It understands templates, overload resolution, SFINAE, concepts, and modules.
- It evaluates code after preprocessing and parsing.
- It can reason about types, inheritance, and instantiation contexts.

In large module-based projects, this semantic precision is critical:

- It avoids textual false alarms common in regex-based linters.
- It respects module boundaries.
- It analyzes exported declarations consistently.

7.1.2 Custom Rule Sets at Scale

Designing a layered rule policy

A professional codebase typically uses layered rule sets:

1. **Foundation layer**: correctness (bugprone, concurrency).
2. **Safety layer**: undefined behavior, lifetime.
3. **Modernization layer**: C++20/23 idioms.
4. **Performance layer**: expensive copies, unnecessary allocations.
5. **Readability layer**: naming, formatting consistency.

Example enterprise-grade configuration

```
Checks: >
  -*,
  bugprone-*,
  performance-*,
  modernize-*,
  concurrency-*,
  cppcoreguidelines-*,
  readability-*,
  -readability-identifier-length,
  -cppcoreguidelines-avoid-magic-numbers

WarningsAsErrors: >
  bugprone-*,
  concurrency-*,
  cppcoreguidelines-interfaces-global-init
```

```
HeaderFilterRegex: '^(src|include)/'

CheckOptions:
  - key:             modernize-use-nullptr.NullMacros
    value:           'NULL'
  - key:             readability-identifier-naming.VariableCase
    value:           lower_case
```

Policy evolution strategy

Large projects rarely enable all checks at once. Instead:

1. Generate baseline report.
2. Freeze legacy warnings.
3. Enforce new code compliance.
4. Gradually reduce technical debt.

7.1.3 Creating Custom clang-tidy Checks

Organizations with domain-specific constraints often write custom AST matchers.
Typical internal checks include:

- Forbid dynamic memory allocation in real-time components.
- Enforce usage of company-specific smart pointer wrappers.
- Forbid direct use of certain unsafe APIs.

Custom checks are implemented using:

- Clang AST matchers.

- Custom diagnostic emitters.
- Automated fix-it hints.

This elevates `clang-tidy` from linting to enforceable architecture governance.

7.1.4 Team-Wide Rule Enforcement

Strict CI enforcement model

A mature enforcement workflow:

1. CI runs clang-tidy against diff.
2. Only modified lines are evaluated (diff-based filtering).
3. New violations fail the build.
4. Suppressions require justification in code review.

Preventing rule drift

Avoid:

- Random NOLINT usage without explanation.
- Disabling warnings globally.
- Divergent rule sets across teams.

Instead:

- Maintain centralized rule ownership.
- Require documentation for suppressed warnings.
- Review suppression patterns periodically.

7.1.5 Auto-Fixing at Scale

Safe vs Unsafe Fixes

Safe fixes:

- `NULL` → `nullptr`
- `typedef` → `using`
- Remove redundant `std::move`

Potentially unsafe:

- Transform raw loops to algorithms.
- Refactor ownership semantics.

Mass Refactoring Pipeline

1. Create modernization branch.
2. Run clang-tidy with `-fix`.
3. Run full test suite.
4. Run static analyzers.
5. Merge incrementally.

7.2 Sanitizers In-Depth (Advanced Engineering)

7.2.1 AddressSanitizer (ASan)

Memory Model Instrumentation

ASan inserts:

- Red zones around allocations.
- Shadow memory tracking.
- Stack frame poisoning.

Real-World Case: Iterator invalidation

```
std::vector<int> v = {1,2,3};
auto it = v.begin();
v.push_back(4);
int x = *it; // use-after-realloc
```

ASan:

- Detects heap-use-after-free.
- Identifies allocation and deallocation site.

Case: Double delete

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

ASan identifies:

- double-free,
- stack trace.

7.2.2 ThreadSanitizer (TSan)

Memory race detection model

TSan tracks:

- Happens-before relationships.
- Atomic operations.
- Lock/unlock sequences.

Real-world race: lazy singleton

```
static MyType* instance = nullptr;

MyType* get() {
  if (!instance)
    instance = new MyType();
  return instance;
}
```

TSan flags:

- data race on pointer.

Correct approach:

```
static MyType instance;
return &instance;
```

or use `std::call_once`.

7.2.3 UndefinedBehaviorSanitizer (UBSan)

Shift overflow

```
int x = 1 << 31; // UB if int is 32-bit signed
```

UBSan reports:

- shift-out-of-bounds.

Misaligned pointer access

```
char buf[4];
int* p = reinterpret_cast<int*>(buf + 1);
int v = *p; // misaligned
```

UBSan:

- detects alignment violation.

7.2.4 Integrating Sanitizers into CI

Multi-configuration matrix

Professional CI typically includes:

- Debug + ASan
- Debug + TSan
- Debug + UBSan
- Release + LTO

CMake pattern

```
option(ENABLE_ASAN "Enable AddressSanitizer" OFF)

if(ENABLE_ASAN)
  target_compile_options(app PRIVATE -fsanitize=address -fno-omit-frame-pointer)
  target_link_options(app PRIVATE -fsanitize=address)
endif()
```

Fail-fast policy

- Sanitizer error = CI failure.
- No suppression without documented reason.

7.3 Static Analysis (Advanced)

7.3.1 False Positives & Noise Reduction

Noise categories

- Template metaprogramming.
- Platform abstraction.
- Defensive programming patterns.

Noise mitigation techniques

- Restrict analysis scope.
- Use targeted suppressions.
- Refactor ambiguous constructs.

7.3.2 Combining Multiple Analyzers

Layered model:

1. Compiler warnings.
2. clang-tidy.
3. Static analyzer.
4. Sanitizers.
5. Security scanners.

Each tool covers different bug classes.

7.3.3 Whole-Program Analysis

Whole-program analysis allows:

- Inter-procedural lifetime tracking.
- Cross-TU resource ownership modeling.
- Global taint propagation.

Modules improve this by:

- Reducing include noise.
- Providing clear symbol boundaries.
- Improving semantic clarity.

7.3.4 Security-Oriented Static Checks

Common vulnerabilities detected

- Buffer overflow.
- Use-after-free.
- Unchecked input.
- Integer overflow.
- Improper synchronization.

Secure coding enforcement

- Prefer RAII.
- Avoid raw owning pointers.
- Use bounds-checked containers.
- Use strong types for identifiers.

7.4 Large-Scale Tooling Architecture

7.4.1 Full Toolchain Pipeline

1. Build generates `compile_commands.json`.
2. clang-tidy runs per commit.
3. Static analyzer runs nightly.

4. Sanitizer builds run on test suites.
5. Security scans run before release.

7.4.2 Metrics for Code Health

- Warning count trend.
- Sanitizer violation trend.
- Static analyzer finding density.
- Mean time to fix tooling-detected bug.

7.5 Engineering Checklist

- No new warnings introduced.
- Sanitizer-clean before tagging release.
- clang-tidy baseline maintained.
- Suppression documented.
- Module boundaries equally analyzed.

7.6 Advanced Exercises

Exercise 1

Introduce subtle UB via signed overflow. Detect with UBSan and refactor using safe arithmetic.

Exercise 2

Simulate race condition in module-based component. Detect with TSan. Refactor with atomic memory ordering discipline.

Exercise 3

Create custom clang-tidy rule that forbids raw `new` in exported module interfaces.

Exercise 4

Configure CI matrix with:

- Clang + ASan
- Clang + TSan
- GCC + UBSan
- MSVC warnings level maximum

Ensure failures block merge.

Exercise 5

Measure tool overhead impact:

- Compile time increase with clang-tidy enabled.
- Runtime overhead with ASan.
- Memory overhead measurement.

Compare Debug vs Release builds.

Part IV

Program Memory Layout & Low-Level Structure

Executable Memory Layout (Cross-Platform)

8.1 Why executable formats matter to C++ engineers

Modern C++ programs are not just "compiled code." They are:

- a structured binary container describing *what* must be mapped into memory and *how*,
- a dynamic linking contract describing *which symbols* must be resolved at load time or on demand,
- a startup protocol describing *what must run before* `main` (global constructors, runtime initialization),
- a runtime metadata layout for exception handling, stack unwinding, RTTI, TLS, profiling, and sanitizers.

Across platforms, the container format differs:

- Linux and many Unix-like systems: **ELF**
- Windows: **PE/COFF**
- macOS: **Mach-O**

But the core questions remain the same:

1. What gets mapped? (segments)
2. What is for link-time bookkeeping only? (sections)
3. How are dependencies loaded? (dynamic loader)
4. How are initializers executed? (global constructors)
5. How is thread-local state realized? (TLS)

8.2 ELF (Linux)

8.2.1 ELF mental model: two orthogonal views

ELF has two main table-driven views of the same file:

- **Sections** (`.text`, `.rodata`, `.data`, `.bss`, `.symtab`, `.dynsym`, `.rela.*`, `.init_array`, `.fini_array`, `...`)
- **Segments** (program headers: `PT_LOAD`, `PT_DYNAMIC`, `PT_INTERP`, `PT_TLS`, `...`)

Rule of thumb:

> Sections are for linkers and tooling. Segments are for loaders and runtime execution.

8.2.2 Segments vs sections

Sections: fine-grained organization

Sections partition the file into logical units used by:

- the static linker (`ld`) during final link,
- the dynamic linker (`ld.so`) when processing dynamic relocations,
- tools such as `readelf`, `objdump`, `nm`, `strip`,
- debuggers and profilers (`.debug_*`, `.eh_frame`, etc.).

Typical sections and their purpose:

- `.text`: executable code (read + execute)
- `.rodata`: read-only constants (read-only)
- `.data`: initialized writable data
- `.bss`: zero-initialized data (no bytes in file; memory is zeroed)
- `.dynsym` / `.dynstr`: dynamic symbol table and strings
- `.rela.dyn`, `.rela.plt` (or `.rel.*`): relocation records
- `.got`, `.plt`: indirection tables for dynamic linking
- `.init_array`, `.fini_array`: constructor/destructor function pointer arrays

Segments: what becomes the process image

Segments are described by the **program header table**. Each entry typically describes:

- file offset + file size,
- virtual address + memory size,
- permissions (R/W/X),

- alignment constraints,
- semantic type (loadable, dynamic, interpreter, TLS, ...).

Most important segment types:

- `PT_LOAD`: a loadable region mapped into memory. Usually multiple such segments exist:
  - one RX for `.text` and some `.rodata`
  - one R for additional rodata
  - one RW for `.data` + `.bss`
- `PT_INTERP`: path to the dynamic loader (interpreter) that will run first for dynamically linked executables
- `PT_DYNAMIC`: points to the `.dynamic` table describing needed libraries, relocation info, symbol tables, and more
- `PT_TLS`: thread-local storage image (template for per-thread data)

Why the loader cares about segments, not sections

The kernel loader (and dynamic loader) maps memory at page granularity with protection flags. Segments provide exactly what the loader needs:

- where to map,
- how many bytes to map from file,
- how many bytes to allocate in memory (including BSS expansion),
- what permissions to enforce.

8.2.3 Dynamic loader (ld.so): the real entry before `main`

For a dynamically linked ELF executable, control flow starts conceptually like this:

1. Kernel loads the ELF, reads program headers.
2. If `PT_INTERP` exists, kernel maps and transfers control to the interpreter (dynamic loader).
3. The dynamic loader:
    - loads required shared objects (from `DT_NEEDED`),
    - performs relocation processing (`REL/RELA`),
    - resolves symbols (eagerly or lazily),
    - prepares TLS, thread-local runtime state,
    - runs initializers,
    - then jumps into the program's startup routine which eventually calls `main`.

Relocations: why addresses are not final in a PIE world

Modern Linux executables are often built as PIE, and shared libraries are position-independent. Therefore:

- load addresses vary due to ASLR,
- many references require relocation fixups at runtime.

Two broad relocation families matter in practice:

- **Non-PLT relocations** (`.rela.dyn`): fix data pointers, vtables, typeinfo pointers, static addresses.
- **PLT/GOT relocations** (`.rela.plt`): support function call indirection for external calls.

Lazy vs immediate binding (conceptual)

- **Immediate binding**: resolve external function symbols during startup (longer startup, simpler runtime behavior).
- **Lazy binding**: resolve functions on first call via PLT trampolines (faster startup, first-call overhead, different failure timing).

8.2.4 Global constructors (Linux): `.init_array` and friends

C++ guarantees that objects with static storage duration are initialized before entering `main` (with well-defined ordering constraints within a TU and partial ordering across TUs).
On ELF systems, these are commonly realized as function pointers collected into:

- `.init_array`: array of constructor function pointers (executed in ascending order)
- `.fini_array`: array of destructor function pointers (executed in reverse order)

Example: global object constructor

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

struct Logger {
  Logger()  { std::puts("Logger::Logger() before main"); }
  ~Logger() { std::puts("Logger::~Logger() after main"); }
};

Logger g_logger;
static std::string g_name = "acme";

int main() {
```

```
  std::puts("main()");
}
```

What this implies in the binary:

- compiler emits a hidden initializer function for `g_logger` and `g_name`,
- that function's address is placed into `.init_array`,
- the runtime (via loader/startup) calls those init functions before `main`.

Constructor attributes (C/C++)

```
#include <cstdio>

__attribute__((constructor))
static void before_main() {
  std::puts("constructor attribute ran");
}
```

This also typically contributes entries to the initializer array.

Architectural warning: initializer order across shared libraries

In large systems:

- dynamic libraries have their own initializer lists,
- initialization order across DSOs depends on dependency order and loader behavior,
- relying on cross-library initialization order is fragile.

Engineering guidance:

- avoid complex work in global constructors,

- prefer explicit initialization functions called from a controlled startup phase,
- keep globals as trivial and constexpr-initialized when possible.

8.3 PE (Windows)

8.3.1 PE mental model: image + loader contract

PE (Portable Executable) is the Windows executable image format. It is tightly integrated with:

- the Windows loader (mapping, relocations, imports),
- the COFF object format used by MSVC toolchains,
- Windows DLL import/export mechanics.

8.3.2 PE structure

High-level layout

A typical PE image contains:

- **DOS header** (legacy stub, `MZ`, and pointer to PE header)
- **PE signature** (`PE\0\0`)
- **COFF file header**
- **optional header** (despite name, it is essential for executables/DLLs):
  - image base,
  - entry point RVA,
  - section alignment,

  - data directories (imports, exports, relocations, TLS, resources, exception data, etc.)

- **section table** (describes `.text`, `.rdata`, `.data`, `.pdata`, `.reloc`, ...)
- **section contents** (raw data mapped into memory)

Sections vs memory protections

Each PE section includes flags describing:

- readability/writability/executability,
- whether it contains code, initialized data, or uninitialized data,
- alignment properties.

The loader maps the image into memory with page-level protections derived from section flags.

8.3.3 Import tables

Why imports are first-class in PE

Windows linking model for DLLs is explicit:

- A module (EXE/DLL) imports symbols from other DLLs.
- The PE import directory describes which DLLs and which symbols are required.
- The loader resolves these imports and patches the Import Address Table (IAT).

Core import structures (conceptual)

The import system revolves around:

- **Import Directory**: list of imported DLLs.

- **Import Name Table (INT)**: names/hints of imported symbols.
- **Import Address Table (IAT)**: actual function pointers used by code after loader fixup.

Call path intuition

After loading:

- external calls are typically made through IAT entries,
- code reads the target address from IAT and jumps/calls it,
- the loader ensures IAT contains resolved addresses for imported functions.

Delay-load (engineering pattern)

Many Windows applications use delay-load semantics for optional features:

- import resolution can occur when the function is first called,
- failures can be handled dynamically (feature fallback),
- startup time may improve by deferring work.

8.3.4 TLS on Windows

Two layers: language TLS and OS TLS

Windows provides TLS support at the OS loader/runtime level, and C++ provides `thread_local`. These interact through compiler and runtime mechanisms.
A useful mental model:

- The PE image may contain a **TLS directory** describing:
  - TLS data template in the image,

  - TLS index used by runtime,
  - optional TLS callbacks.

- Each thread gets its own instance of TLS data.

TLS callbacks

Windows supports **TLS callbacks**: functions called by the loader on:

- process attach/detach,
- thread attach/detach.

This is distinct from C++ global constructors and is often used by runtimes and special libraries.

C++ `thread_local` with non-trivial constructors

Non-trivial `thread_local` objects require per-thread initialization and destruction.

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

thread_local std::string tls_name = "worker";

static void touch() {
  std::puts(tls_name.c_str());
}

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

Engineering consequences:

- each thread may run initialization code the first time it uses the variable (or at thread start, depending on implementation),
- cleanup occurs at thread exit for variables with destructors,
- heavy TLS objects can become a scalability cost in thread-heavy programs.

8.3.5 Global constructors on Windows (CRT initialization)

C++ global constructors are orchestrated by the C/C++ runtime startup, not directly by the PE format itself. However, PE provides places to store initializer lists and metadata that the runtime uses.
Engineering viewpoint:

- The OS loader maps the image and resolves imports/relocations.
- Then the entry point transitions into runtime startup.
- Runtime startup runs global initializers, then calls `main` (or `WinMain`), and later runs destructors.

The existence of global constructors implies:

- a phase boundary before `main`,
- potential ordering hazards across DLL boundaries,
- the importance of controlling initialization in large architectures.

8.4 Mach-O (macOS)

8.4.1 Mach-O mental model: load commands drive everything

Mach-O binaries are structured around:

- a header,
- **load commands** (the real "table of contents"),
- segments and sections mapped into memory.

Load commands specify:

- which dynamic linker to use,
- which dynamic libraries are required,
- where segments live and how to map them,
- symbol tables and dynamic linking metadata,
- code signing information,
- runtime paths and versions.

8.4.2 Segments & pages

Segments and sections

Mach-O groups data into **segments** (often page-aligned) that contain **sections**. Typical naming conventions include:

- `__TEXT` segment: code and read-only data
- `__DATA` segment: writable data
- `__LINKEDIT` segment: linking metadata (symbol tables, strings, binding info)

Common sections include:

- `__TEXT,__text`: executable code

- `__TEXT,__cstring`: constant C strings
- `__DATA,__data`: initialized data
- `__DATA,__bss`: zero-initialized data
- `__DATA,__mod_init_func`: initializer function pointers (global constructors and similar)

Pages and locality

macOS paging behavior makes code/data locality a performance concern:

- segments are mapped with page granularity,
- hot code clustered into fewer pages reduces page faults and improves startup,
- cold code separated reduces working set pressure.

8.4.3 Dyld

Dyld as the runtime linker and binder

`dyld` is responsible for:

- loading dependent dynamic libraries,
- performing rebasing (ASLR adjustments),
- binding symbols (resolving imports),
- running initializers,
- transferring control to the program entry point and runtime startup.

Rebasing vs binding (conceptual)

- **Rebasing**: adjusting pointers that assume a preferred load address when the image is loaded elsewhere.
- **Binding**: resolving references to external symbols (functions/objects) from other images.

8.4.4 Library loading

Load commands and dependency declarations

Mach-O records library dependencies via load commands. At runtime:

- dyld locates libraries using configured search paths and embedded path metadata,
- maps them into memory,
- resolves symbol references,
- runs initializer routines.

Initializer execution: `__mod_init_func`

Mach-O commonly stores initializer function pointers in `__DATA,__mod_init_func`. dyld runs these before `main`:

- C++ global object constructors,
- functions marked with constructor attributes,
- Objective-C class loading behaviors (in mixed-language programs).

Example: global constructor behavior is format-driven but language-owned

```
#include <cstdio>

struct X {
  X()  { std::puts("X() before main on macOS too"); }
  ~X() { std::puts("~X() after main on macOS too"); }
};

X gx;

int main() {
  std::puts("main()");
}
```

Same C++ semantics, different container mechanics:

- ELF typically uses `.init_array`.
- Mach-O typically uses `__mod_init_func`.
- PE relies on runtime conventions supported by PE directories/sections and CRT startup.

8.5 Cross-platform comparisons that matter in real engineering

8.5.1 Segments vs sections (unified perspective)

Across formats:

- ELF: runtime mapping uses **segments** (program headers), tooling uses **sections**.
- Mach-O: runtime mapping uses **segments** described by **load commands**; sections exist within segments for organization and tooling.

- PE: runtime mapping uses **sections** with flags and alignment; directories provide structured metadata for loader tasks.

8.5.2 Dynamic loader responsibilities (unified perspective)

- Linux (`ld.so`): load DSOs, relocate, bind symbols (lazy/eager), run init arrays.
- Windows loader: map image, apply relocations, resolve imports into IAT, run TLS callbacks; CRT then runs C++ initializers.
- macOS (`dyld`): map images, rebase, bind, run initializers, transition into runtime startup.

8.5.3 Global constructors: why they are an architectural hazard

Global constructors are convenient but dangerous at scale:

- hidden work before `main` (startup time surprises),
- implicit ordering constraints across binaries/libraries,
- failure handling is awkward (exceptions, abort paths),
- concurrency issues when initialization meets threads (especially with plugins).

Large-system discipline:

- keep static initialization trivial,
- prefer `constexpr` initialization where possible,
- centralize initialization in a dedicated startup layer,
- treat dynamic loading and plugin initialization as explicit protocols.

8.6 Practical diagnostics & experiments (hands-on)

These experiments build intuition without needing platform-specific internals in your code.

Experiment 1 (Linux): map view vs tool view

- Compare program headers vs sections (segments vs sections).
- Identify which regions are RX, R, RW.
- Locate initializer arrays and dynamic linking metadata.

Experiment 2 (Windows): imports and TLS

- Inspect which DLLs are imported and where the IAT is located.
- Observe how delay-load changes startup behavior.
- Inspect the TLS directory and see how TLS callbacks appear (if present).

Experiment 3 (macOS): load commands and initializers

- Examine load commands to see which dylibs are required.
- Locate `__mod_init_func` and correlate with global constructors.
- Observe how rebasing/binding affects addresses under ASLR.

8.7 Engineering checklist (Chapter 8)

1. Can you explain which parts of your binary are mapped RX, R, RW, and why?

2. Do you understand where global constructors live in your platform format?

3. Do you know when imports are resolved (eager vs lazy vs delay-load)?

4. Do you understand how TLS is represented and what it costs?

5. Can you keep startup work explicit rather than hidden in global initialization?

Stack & Heap Engineering

9.1 Stack Frame Internals

9.1.1 Why stack engineering still matters in Modern C++

Even with modules, LTO, and high-level abstractions, the machine still executes a calling convention, builds stack frames, preserves registers, and follows unwind rules for exceptions, debugging, and profiling.
In large systems, misunderstanding stack mechanics causes:

- ABI breaks across shared libraries,
- crashes in exception unwinding and stack walking,
- performance regressions (missed tail calls, excess spills),
- undefined behavior when mixing compilers/flags/assembly.

9.1.2 Stack frame anatomy (unified model)

A stack frame is the region of memory used by a function activation. Across platforms, you usually see these components:

- **Return address**: where control resumes after the call.

- **Caller argument spill / home area**: stack space reserved for passing arguments beyond registers or for ABI-required homes.
- **Saved registers**: callee-saved registers preserved across the call.
- **Local variables**: scalars, arrays, aggregates, and temporaries.
- **Outgoing call argument area**: stack space used for calls made by the function.
- **Alignment padding**: to satisfy ABI stack alignment.
- **Unwind metadata**: not stored *inside* the frame, but describing how to unwind it (DWARF CFI on ELF; UNWIND_INFO on Windows; similar metadata on macOS).

9.1.3 Call ABI differences

A serious cross-platform engineer must treat ABI as a contract with these axes:

- **Argument passing**: which registers, when stack, how aggregates are lowered.
- **Return values**: which registers, sret conventions for large structs.
- **Callee-saved vs caller-saved**: register volatility rules.
- **Stack alignment**: required alignment at call boundaries.
- **Prologue/epilogue constraints**: how stack pointer and frame pointer are used.
- **Exception/unwind metadata format**: how stack walking and unwinding are represented.

System V AMD64 ABI (Linux, many Unix-like x86-64 environments)

High-level essentials:

- Integer/pointer arguments typically start in registers (commonly: RDI, RSI, RDX, RCX, R8, R9), remaining passed on stack.
- Return values typically in RAX (and RDX for larger integer pairs).
- Stack grows downward; stack pointer is RSP.
- Stack alignment is enforced at call boundaries (the ABI requires a specific alignment rule; compilers maintain it).
- A special concept exists: **red zone** (a small region below RSP that leaf functions may use without adjusting RSP).

Engineering consequences:

- Leaf functions may avoid allocating stack by using red-zone space, improving performance.
- Signal/interrupt handling and kernel transitions are relevant to correctness assumptions; compilers generate code according to the ABI contract for user space.

Windows x64 ABI (MSVC toolchain, Windows)

High-level essentials:

- First integer/pointer args typically in RCX, RDX, R8, R9; additional args on stack.
- **Shadow space (home space)**: the caller reserves a fixed stack area for the callee to spill the register arguments if needed.

- Stack alignment is strictly maintained.
- Exception handling and stack unwinding are tightly coupled to explicit unwind metadata (UNWIND_INFO).

Engineering consequences:

- The presence of shadow space changes prologue patterns and affects handwritten assembly.
- Unwind metadata is mandatory for reliable stack walking and exceptions; inline assembly is constrained compared to other platforms.

AArch64 ABI (Linux/macOS on ARM64, many platforms)

High-level essentials:

- Many arguments pass in X0–X7 (integer/pointer) and V0–V7 (floating/SIMD) with stack fallback.
- Return values typically in X0 (and possibly X1 for pairs or large returns via hidden pointer).
- Link register (LR / X30) holds return address; calls use BL/BLR and returns use RET.
- Frame pointer is often X29 when used.
- Stack pointer alignment and unwind rules are ABI-defined; compilers follow strict rules for debuggability and EH.

Engineering consequences:

- The return address is not pushed automatically by hardware; prologue often saves LR if the function is non-leaf.
- Proper unwind metadata and consistent prologue/epilogue patterns are crucial for reliable unwinding/backtraces.

9.1.4 Register saving & unwind rules

The volatility contract

Every ABI splits registers into:

- **Caller-saved (volatile)**: call may clobber; caller preserves if needed.
- **Callee-saved (non-volatile)**: callee must restore to preserve caller state.

The compiler's register allocator relies on this contract:

- It prefers volatile registers for short-lived values.
- It saves/restores non-volatile registers only when needed (spill cost).

Unwinding and stack walking: why metadata exists

Unwinding and stack walking require answering:

- Where is the caller's return address?
- What is the caller's stack pointer value?
- Which callee-saved registers were pushed, and where?

This cannot be reliably inferred from machine code in optimized builds because:

- frame pointers may be omitted,
- prologues can be optimized (shrink-wrapping),
- registers may be saved conditionally,
- tail calls can remove frames.

Therefore platforms use explicit unwind metadata:

- **DWARF CFI** (common on ELF and often used in macOS tooling too): describes how to recover the caller context at each instruction range (CFA rules).
- **Windows x64 UNWIND_INFO**: compact encoding of prologue actions (stack allocation, register saves) used by SEH and stack walking.

Practical C++: how prologues appear

A typical non-leaf function (conceptually) does:

1. allocate stack space (adjust SP),
2. save callee-saved registers it will use,
3. establish a frame pointer (optional, depends on optimization/debug policy),
4. run body,
5. restore registers,
6. deallocate stack space,
7. return.

Example: forcing a non-trivial stack frame in C++

```
#include <cstdint>
#include <array>

static volatile std::uint64_t sink = 0;

std::uint64_t heavy_stack(std::uint64_t x) {
```

```
  std::array<std::uint64_t, 64> buf{};  // forces stack usage
  for (std::size_t i = 0; i < buf.size(); ++i) {
    buf[i] = x + i;
  }

  std::uint64_t sum = 0;
  for (auto v : buf) sum += v;

  sink = sum; // prevent optimization removing everything
  return sum;
}
```

This tends to create:

- local stack allocation,
- register pressure (possible spills),
- clear unwind metadata.

Frame pointer omission and debugging trade-offs

In many toolchains, you can choose policies:

- **Keep frame pointers**: easier profiling/backtraces, slightly less optimization freedom.
- **Omit frame pointers**: frees a register, can improve performance, but stack walking relies fully on metadata.

In production-grade systems:

- You often keep frame pointers in server builds for observability.
- You may omit in hot performance builds where instrumentation is less needed.

9.1.5 Tail calls

Tail call: definition

A **tail call** happens when a function returns the result of calling another function as its final action. If legal and profitable, the compiler can transform it into a jump, reusing the current stack frame (**tail call elimination**).

Why tail calls matter

- **Performance**: eliminates call/return overhead, reduces stack churn.
- **Correctness in recursion**: can prevent stack overflow for tail-recursive patterns (but C++ does not guarantee this optimization).
- **Unwinding/backtraces**: tail calls can remove frames, changing stack traces.

When tail calls are blocked

Common blockers:

- needing to run destructors for local objects after the call,
- calling conventions mismatch (ABI constraints),
- variadic calls or special calling conventions,
- taking the address of a local that must remain valid,
- exceptions and EH regions that require cleanup after the call,
- debug/profiling instrumentation that requires full call frames.

Example: tail call blocked by destructor

```
#include <string>

int g(int);

int f(int x) {
  std::string s = "work"; // destructor must run at end of scope
  return g(x);            // tail call typically blocked because s must be destroyed
}
```

Example: tail call more likely when no cleanups exist

```
int g(int);

int f(int x) {
  return g(x); // tail call candidate
}
```

Engineering note: tail calls and ABI

Tail calls are ABI-sensitive because the caller's outgoing argument area and stack alignment must remain valid for the jump. On Windows x64, the presence of shadow space and unwind metadata constraints can influence how tail calls are realized. On AArch64, tail calls are often straightforward but still depend on register usage and required cleanups.

9.2 Heap Internals

9.2.1 The heap is not one thing

When engineers say "the heap", they usually mean a combination of:

- the C runtime allocator interface (`malloc/free`),
- the C++ dynamic allocation interface (`operator new/delete`),
- allocator strategies inside containers (`std::allocator`, custom allocators),
- the OS virtual memory subsystem backing it (mmap/VirtualAlloc and page management),
- optional alternative allocators (tcmalloc/jemalloc/mimalloc-like strategies).

A professional C++ engineer must separate:

- **API level**: what your code calls (`new`, `malloc`, PMR).
- **Implementation level**: how memory is carved into blocks, cached, aligned, and returned.
- **Workload level**: allocation sizes, lifetimes, thread patterns, locality requirements.

9.2.2 malloc, new, allocators

C: `malloc/free`

Properties:

- allocates raw bytes, returns `void*`.
- no constructors/destructors.
- alignment guarantees follow the platform contract (suitable for any object type up to a certain alignment).

C++: `operator new/delete`

Properties:

- `new T` typically does: allocate raw storage + construct `T`.
- `delete p` typically does: destruct `T` + deallocate storage.
- `operator new` throws `std::bad_alloc` by default on failure (unless `nothrow` form is used).

Allocators for containers

Containers allocate internal storage via allocator-aware mechanisms. This enables:

- custom allocation strategies without changing container logic,
- separation of data structure logic from allocation policy,
- memory locality optimizations (arena allocation),
- control of fragmentation and per-thread caching.

Example: observe allocation behavior by overriding global new/delete

This technique is used for diagnostics (not always recommended in production libraries):

```
#include <new>
#include <cstdio>
#include <cstdlib>

void* operator new(std::size_t n) {
  std::printf("[new ] %zu bytes\n", n);
  if (void* p = std::malloc(n)) return p;
  throw std::bad_alloc{};
```

```
}

void operator delete(void* p) noexcept {
  std::printf("[del ] %p\n", p);
  std::free(p);
}
```

Alignment-aware allocation

Modern C++ supports over-aligned types. Allocation must respect alignment:

```
#include <cstddef>
#include <new>
#include <iostream>

struct alignas(64) CacheLine {
  std::byte data[64];
};

int main() {
  CacheLine* p = new CacheLine;
  std::cout << "ptr=" << static_cast<void*>(p) << "\n";
  delete p;
}
```

Engineering note:

- over-aligned allocations may route to special aligned allocation paths,
- mixing allocators that do not respect alignment is a correctness bug.

9.2.3 Fragmentation control

What fragmentation actually means

Two major fragmentation types:

- **External fragmentation**: free memory exists, but split into chunks too small/too scattered for a large request.
- **Internal fragmentation**: allocator returns a block larger than requested (due to size class rounding, headers, alignment).

Large codebases often suffer fragmentation because:

- allocation sizes vary widely,
- lifetimes vary widely,
- multi-threaded allocation patterns create contention and cache overhead,
- long-running services accumulate allocation "shape" that never fully returns to a compact state.

Size classes and pooling

Many allocators reduce fragmentation and improve speed using:

- **size classes**: allocate from bins (e.g., 16, 32, 64, 128 bytes, etc.).
- **thread caches**: per-thread free lists for small allocations.
- **arenas**: separate allocation domains to reduce lock contention.

This often increases internal fragmentation but reduces external fragmentation and improves throughput.

Lifetime segregation: the most powerful fragmentation technique

A practical rule:

> Allocate objects with similar lifetimes from the same allocator/arena.

Examples:

- request-scoped allocations (freed at end of request),
- frame-scoped allocations (freed at end of game frame),
- compilation unit pass allocations (freed at end of compilation pass),
- message batch allocations (freed when batch is processed).

This creates predictable deallocation patterns and can reduce fragmentation dramatically.

Arenas in Modern C++ using PMR

Polymorphic allocators enable an arena style without rewriting containers:

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

struct Request {
  std::pmr::vector<std::pmr::string> tokens;
};

Request parse_request(std::pmr::memory_resource* mr) {
  Request r{ std::pmr::vector<std::pmr::string>(mr) };
  r.tokens.emplace_back("GET", mr);
  r.tokens.emplace_back("/api/items", mr);
```

```
  r.tokens.emplace_back("HTTP/1.1", mr);
  return r; // tokens stored in the request arena
}

int main() {
  std::byte buffer[4096];
  std::pmr::monotonic_buffer_resource arena(buffer, sizeof(buffer));

  Request r = parse_request(&arena);
  for (auto& s : r.tokens) std::cout << s << "\n";

  // All memory released together when arena is destroyed
}
```

Properties:

- extremely fast allocations,
- very low per-allocation overhead,
- no per-object frees (free-all-at-once).

When it is appropriate:

- request pipelines,
- parsing/tokenization,
- transient graphs,
- compiler passes,
- batch processing.

When it is inappropriate:

- long-lived objects with independent lifetimes,
- allocations that must be freed individually,
- unbounded growth without reset (memory retention risk).

9.2.4 Alignment

Alignment constraints: what engineers must guarantee

Alignment affects:

- correctness (misaligned access can be UB or a hardware fault on some architectures),
- performance (cache line splits, unaligned vector loads),
- ABI (struct layout, calling convention, stack alignment).

Stack alignment

Every ABI specifies stack alignment at call sites. A compiler ensures alignment for:

- local variables (including over-aligned objects),
- vectorized code that requires aligned stack data,
- variadic argument passing rules (platform dependent).

Heap alignment and over-aligned types

Modern C++ supports `alignas(N)` on types. Your allocator strategy must provide correct alignment, including for:

- SIMD types,

- cache-line aligned node allocations,
- lock-free structures where alignment affects atomicity and false sharing.

A safe aligned allocation pattern (portable C++ style)

```
#include <cstdlib>
#include <new>
#include <cstddef>

struct alignas(64) Node { int x; };

Node* make_node() {
  return new Node{42}; // relies on aligned operator new for over-aligned Node
}

void destroy_node(Node* p) {
  delete p;
}
```

If you implement custom allocation for over-aligned types, you must ensure:

- correct aligned allocation and deallocation pairing,
- no mixing of allocation APIs with incompatible alignment semantics.

9.2.5 Custom allocator strategy

Allocator strategy is architecture

In large systems, allocator strategy is not a micro-optimization; it becomes a structural design choice. A good strategy starts from workload analysis:

- allocation sizes distribution,

- allocation lifetime distribution,
- thread contention patterns,
- locality requirements (NUMA, cache lines),
- failure behavior (must never throw, must be bounded, must be deterministic).

Common allocator archetypes

1) Monotonic / arena allocator (bump pointer)

Best for:

- bulk free (free all at once),
- request lifetime,
- parsing, AST building, temporary graphs.

Trade-offs:

- cannot free individual objects,
- can retain memory until reset/destroy,
- requires careful lifetime design.

2) Pool allocator (fixed-size blocks)

Best for:

- many allocations of the same size (nodes, messages),
- stable latency and predictable performance,

- reducing fragmentation for small objects.

Trade-offs:

- poor for variable-sized allocations,
- requires size-class strategy or multiple pools.

3) Segregated fits / size-class allocators (general purpose)

Best for:

- general-purpose allocation across many sizes,
- multi-threaded throughput,
- long-running services.

Trade-offs:

- more complex implementation,
- internal fragmentation due to rounding and metadata overhead.

4) Region-per-thread / thread caching

Best for:

- high concurrency workloads,
- many small allocations in hot paths.

Trade-offs:

- memory can accumulate in thread caches,
- cross-thread free patterns can be costly without careful design.

Professional pattern: request arena + long-lived heap

A robust architecture commonly uses:

- an arena (PMR monotonic) for request-scoped objects,
- the normal heap (or a tuned allocator) for long-lived objects,
- a pool for frequently allocated node types.

Example: PMR pool on top of a monotonic upstream

This provides:

- fast, low-fragmentation small allocations,
- upstream bulk buffer,
- deterministic lifetime boundary at arena destruction.

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

int main() {
  std::byte upstream_buf[1 << 16];
  std::pmr::monotonic_buffer_resource upstream(upstream_buf, sizeof(upstream_buf));

  std::pmr::unsynchronized_pool_resource pool(&upstream);

  std::pmr::vector<std::pmr::string> v{ &pool };
  for (int i = 0; i < 1000; ++i) {
    v.emplace_back("token", &pool);
  }
```

```
  std::cout << v.size() << "\n";
  // pool and upstream destroyed here -> bulk free
}
```

Engineering notes:

- `unsynchronized_pool_resource` is not thread-safe; use it per-thread or protect it.
- A pool resource reduces fragmentation for many small allocations by pooling blocks.
- Upstream monotonic resource ensures allocation is fast and freed as a whole.

Custom new/delete for a specific type (type-local pool)

Sometimes you want a pool only for one node type:

```
#include <cstddef>
#include <new>
#include <vector>
#include <mutex>

class NodePool {
public:
  void* allocate(std::size_t n) {
    std::lock_guard<std::mutex> lock(m_);
    if (!free_.empty()) {
      void* p = free_.back();
      free_.pop_back();
      return p;
    }
    return ::operator new(n);
  }
```

```
  void deallocate(void* p) noexcept {
    std::lock_guard<std::mutex> lock(m_);
    free_.push_back(p);
  }

  static NodePool& instance() {
    static NodePool pool;
    return pool;
  }

private:
  std::mutex m_;
  std::vector<void*> free_;
};

struct Node {
  int a;
  int b;

  static void* operator new(std::size_t n) {
    return NodePool::instance().allocate(n);
  }

  static void operator delete(void* p) noexcept {
    NodePool::instance().deallocate(p);
  }
};
```

This pattern:

- can reduce fragmentation for node-heavy graphs,
- can improve throughput in allocation hot spots,
- must be engineered carefully for thread contention and memory growth.

Allocator strategy checklist

Before adopting any custom allocator, answer these questions:

1. What allocation sizes dominate? (small nodes, medium strings, large buffers)
2. What are object lifetimes? (per-request, per-session, global)
3. Is allocation on the hot path? (per packet, per frame, per event)
4. Is the workload multi-threaded? (how many threads, what contention)
5. Are there real-time or latency constraints? (bounded time, no page faults)
6. Is memory usage bounded? (risk of arena growth, cache retention)
7. How will you test? (ASan/TSan/UBSan, fuzzing, stress, leak checks)

9.3 Stack & Heap interactions (the part most teams miss)

9.3.1 Escape analysis by humans: when stack allocation is impossible

Many developers hope the compiler "puts things on the stack," but the rule is simple:

> If an object must outlive the function scope, it cannot live on that function's stack.

Common escape causes:

- returning a pointer/reference to a local,
- storing a pointer to local into a global or heap structure,
- capturing a local by reference into an asynchronous task,
- coroutine frames and suspension (many locals become part of the coroutine frame which is typically heap-allocated unless customized).

9.3.2 Tail calls vs heap allocations in recursive designs

Tail call elimination can reduce stack growth, but:

- it is not guaranteed in C++,
- destructors and EH often block it,
- recursion-heavy designs may be better modeled with explicit stacks (heap or arena) for predictability.

9.4 Practical labs (what to do in your own toolchain)

Lab 1: Observe stack frames under optimization

1. Compile a file with `-O0` and `-O2`.
2. Inspect prologue/epilogue differences.
3. Observe frame pointer usage and register spills.

Lab 2: Trigger and analyze a tail call

1. Write a tail-recursive function without local destructors.
2. Compile with high optimization.
3. Inspect assembly to confirm whether tail call elimination occurred.

Lab 3: Fragmentation experiment

1. Allocate many small objects with varied lifetimes using default allocator.

2. Repeat using PMR arena-per-request.
3. Compare peak memory and throughput under stress.

9.5 Engineering checklist (Chapter 9)

1. You can describe your platform calling convention well enough to write correct interop code.
2. You understand which registers are volatile vs non-volatile for your ABI.
3. You know how unwind metadata affects debugging and exceptions.
4. You can explain when tail calls are possible and why they might disappear.
5. You can classify fragmentation type (internal vs external) in a memory profile.
6. You can choose an allocator strategy based on workload, not style.
7. You keep alignment requirements explicit for over-aligned and SIMD-heavy code.

Part V

Full Project: Rebuilding a Real Application Using Modules

Extracting, Refactoring & Designing the Program

10.1 Creating Module Boundaries

10.1.1 Core Principle: Boundaries Are About *Stability*, Not Files

A module boundary is not "whatever fits into one file". A module boundary is the smallest stable contract that:

- changes **less often** than the implementation,
- is **safe to import widely** without leaking internal details,
- and does not rely on **preprocessor state** or include-order accidents.

When refactoring a real application into modules, treat boundaries as **product surfaces**:

- **Domain boundary:** types and rules that define correctness.
- **Service boundary:** operations that use domain types to do work (storage, networking, orchestration).
- **Infrastructure boundary:** platform/OS integrations, third-party libraries, logging, file IO.

- **UI boundary:** presentation and interaction, ideally consuming services, not infrastructure.

A healthy boundary is one that prevents a consumer from accidentally pulling in half the program.

10.1.2 A Practical Boundary Checklist

Before declaring a named module, verify:

- **The public surface is small.** If you must export dozens of unrelated types, the boundary is wrong.
- **No macro requirements.** Public API should not require consumers to define macros in a particular order.
- **No include-order coupling.** Any dependence on "include X before Y" is a refactoring target.
- **Few transitive imports.** If importing one module forces importing many others, design a better facade.
- **Dependency direction is obvious.** Domain should not import UI; infra should not define domain rules.

10.1.3 Choose a Migration Strategy: Three Tracks

In real codebases, a single strategy rarely works. Use three tracks in parallel:

- **Track A: Native named modules (target state).** Best for stable APIs and clean layering.
- **Track B: Header units (bridge).** Useful to reduce include cost and isolate legacy headers while you refactor.

- **Track C: Legacy headers remain.** Some headers will stay as headers until macro-heavy or platform-specific sections are redesigned.

Your goal is to continuously move code from C → B → A.

10.1.4 Boundary Patterns That Scale

Pattern 1: Domain-First Modules

Create a small domain module that exports only the types and invariants that everything else needs.

```
export module app.domain;

export namespace app::domain {

  struct UserId {
    unsigned long long value{};
  };

  struct Money {
    long long cents{};
  };

  enum class Currency { USD, EUR, SAR };

  struct Price {
    Money amount;
    Currency currency;
  };

} // namespace app::domain
```

Everything else imports the domain, but the domain imports almost nothing.

Pattern 2: Interface/Implementation Separation Using Partitions

Keep the interface narrow, push heavy templates/private helpers into partitions.

```
// file: app.storage.ixx
export module app.storage;

import app.domain;

export namespace app::storage {

  struct StorageConfig {
    int cache_mb{};
  };

  class Repository {
  public:
    explicit Repository(StorageConfig cfg);
    void save_user(app::domain::UserId id);
  private:
    struct Impl;
    Impl* p_; // (example: raw pointer for brevity; use smart pointers in real code)
  };

}
```

```
// file: app.storage.impl.ixx  (implementation partition - not exported)
module app.storage:impl;

import app.domain;

namespace app::storage {

  struct Repository::Impl {
```

```
    StorageConfig cfg;
    // heavy includes go here if needed (legacy headers, vendor SDK headers, etc.)
  };

}
```

```
// file: app.storage.cpp  (module implementation unit)
module app.storage;

import :impl;

namespace app::storage {

  Repository::Repository(StorageConfig cfg)
    : p_(new Impl{cfg}) {}

  void Repository::save_user(app::domain::UserId id) {
    (void)id;
    // ...
  }

}
```

This pattern keeps imports stable, limits rebuilds, and stops implementation details from leaking.

Pattern 3: Facade Modules to Control Import Flood

If consumers keep importing many modules just to do one thing, provide a facade module that exports a curated surface.

```
// file: app.facade.ixx
export module app.facade;

export import app.domain;
```

```
export import app.storage;
// export import app.net;    (add only what you truly want re-exported)
// export import app.logging;

export namespace app {
  // Optional: add thin convenience APIs here without exposing internals
}
```

Consumers import `app.facade` and get a stable, intentional set of exports.

10.1.5 When a "Module" Should Actually Be Multiple Modules

Split a module if:

- a subset of consumers needs only 10% of the API but imports 100%,
- the module mixes responsibilities (domain + persistence + UI),
- or the module has conflicting dependency needs (some code must depend on OS headers, other code must not).

Conversely, merge modules if:

- two modules are mutually dependent and cannot be separated without unnatural contortions,
- or they model one concept that was only split due to historical file layout.

10.1.6 A Boundary Mapping Table You Can Use During Refactoring

Boundary Type	Imports Allowed	Forbidden Imports
Domain	minimal C++ stdlib, basic utilities	UI, DB drivers, OS APIs, networking
Services	Domain, narrow infra interfaces	UI concrete types, vendor SDK header
Infrastructure	OS APIs, vendor SDKs, C libraries	Domain rules (avoid coupling correctne to platform)
UI	Services, Domain	Infrastructure details (prefer abstra service APIs)

10.2 Designing Exported Interfaces

10.2.1 Exported Interface Rules for Real Projects

A module interface should be written as if it is your library's public header *for the next five years*.

- Export **types and functions that form a contract**.
- Avoid exporting **implementation-shaped types** (helpers, caches, internal containers).
- Prefer exporting **opaque handles** or **abstract protocols** over concrete subsystems.
- Avoid exporting **macros**; design without requiring them across boundaries.
- Keep exported templates deliberate; exported templates are part of your ABI-and-build story.

10.2.2 Export Only What You Intend to Support

A strong design technique is to treat export as a whitelist.

```
export module app.text;

export namespace app::text {

  export struct Utf8View {
    const char* data{};
    unsigned long long size{};
  };

  export bool is_valid_utf8(Utf8View v);

  // Not exported: internal decoding helpers, tables, caches
  namespace detail {
    bool slow_path_validate(Utf8View v);
  }

}
```

Even if `detail` is in the module interface file, do not export it. Keep your contract surface intentional.

10.2.3 Design for Low Coupling: “Types In, Types Out”

The easiest way to create import cycles is to let each subsystem accept and return other subsystems.
Prefer function signatures that depend on:

- domain types,

- standard vocabulary types,
- and small interfaces (abstract classes, type-erased callables) rather than concrete services.

```
export module app.audit;

import app.domain;

export namespace app::audit {

  export struct AuditEvent {
    app::domain::UserId user;
    int action_code{};
  };

  export class Sink {
  public:
    virtual ~Sink() = default;
    virtual void write(const AuditEvent& e) = 0;
  };

}
```

Now any implementation (file, DB, remote collector) can be in other modules without pulling them into the audit contract.

10.2.4 Use “PImpl” as a Boundary Tool, Not a Habit

The PImpl idiom becomes particularly powerful in modules refactoring because it:

- minimizes exported dependencies,
- keeps rebuild impact low,

- allows heavy legacy headers to remain internal.

```
// file: app.net.ixx
export module app.net;

export namespace app::net {

  export struct Endpoint {
    const char* host{};
    unsigned short port{};
  };

  export class Client {
  public:
    Client();
    ~Client();
    bool connect(Endpoint ep);
  private:
    struct Impl;
    Impl* p_;
  };

}
```

```
// file: app.net.cpp
module app.net;

// heavy legacy headers stay here, not exported
// #include <winsock2.h>
// #include <sys/socket.h>

namespace app::net {

  struct Client::Impl {
```

```
    int dummy{};
  };

  Client::Client() : p_(new Impl{}) {}
  Client::~Client() { delete p_; }

  bool Client::connect(Endpoint ep) {
    (void)ep;
    return true;
  }

}
```

10.2.5 Avoid “Exporting Convenience” That Turns Into Permanent Debt

Common mistakes that become expensive later:

- exporting wide utility headers “just because everyone uses them”,
- exporting a module that re-exports dozens of other modules (uncontrolled transitivity),
- exporting internal containers or third-party types directly,
- exporting configuration structs that expose platform-specific fields.

Instead:

- export narrow vocabulary types you own,
- keep transitive export deliberate via a facade module,
- adapt third-party types behind an interface layer.

10.2.6 Interface Granularity: “One Concept, One Import”

If users must import multiple modules to accomplish one conceptual operation, your boundary is too fragmented. Conversely, if importing one module brings in unrelated concepts, your boundary is too fat.
A useful tactic is to enforce:

- **one primary concept per module** (domain.customer, storage.repository, net.client),
- plus a **single facade** for convenience (app.facade).

10.3 Eliminating Cyclic Imports

10.3.1 Why Cycles Become Non-Negotiable With Modules

With textual headers, cycles are often hidden behind forward declarations, include guards, and include-order coincidences. With named modules, cycles are architectural facts. If module A must import B and B must import A, you do not have two modules; you have one concept that was split incorrectly.
The refactoring goal is not “trick the compiler”. The goal is to redesign the contracts so the dependency graph becomes a DAG.

10.3.2 Cycle Taxonomy: Identify Which Kind You Have

Most cycles fall into one of these buckets:

- **Type cycle:** A exports a type containing B, and B exports a type containing A.
- **Service cycle:** A needs to call B, and B needs to call A (mutual orchestration).
- **Convenience cycle:** exports and helpers got mixed; neither truly needs the other.

- **Config/macro cycle:** both depend on a shared config header or macro state.

Each bucket has a different fix.

10.3.3 Technique 1: Break Type Cycles With Opaque Handles

If two exported types embed each other, stop exporting the concrete ownership relationship. Export an opaque handle or pointer and keep the definition internal.

```
// file: app.a.ixx
export module app.a;

export namespace app {

  export class B; // forward declaration is fine for pointers/references

  export class A {
  public:
    A();
    void attach(B* b);
  private:
    B* b_{};
  };

}
```

```
// file: app.b.ixx
export module app.b;

export namespace app {

  export class A;
```

```
  export class B {
  public:
    void attach(A* a);
  private:
    A* a_{};
  };

}
```

This removes the need for either module interface to import the other. The implementation files can import both modules and define behavior.

```
// file: app.ab.impl.cpp
module app.a;
import app.b;

namespace app {
  A::A() = default;
  void A::attach(B* b) { b_ = b; }
}
```

```
// file: app.ba.impl.cpp
module app.b;
import app.a;

namespace app {
  void B::attach(A* a) { a_ = a; }
}
```

10.3.4 Technique 2: Extract a Shared Vocabulary Module

If the cycle exists because both sides depend on shared enums, IDs, small structs, move those into a third module:

```
// file: app.vocab.ixx
export module app.vocab;

export namespace app {
  export enum class EventKind { Created, Updated, Deleted };
  export struct EntityId { unsigned long long value{}; };
}
```

Then both modules import `app.vocab` instead of importing each other.

10.3.5 Technique 3: Replace Mutual Calls With Inversion of Control

Service cycles often occur when modules both orchestrate each other. The fix is usually to invert control by exporting an interface from one side and injecting an implementation from the other.

```
// file: app.scheduler.ixx
export module app.scheduler;

export namespace app {

  export class Job {
  public:
    virtual ~Job() = default;
    virtual void run() = 0;
  };

  export class Scheduler {
  public:
    void submit(Job& j);
  };

}
```

```
// file: app.scheduler.cpp
```

```
module app.scheduler;

namespace app {
  void Scheduler::submit(Job& j) { j.run(); }
}
```

Now a module that *previously* needed to import the scheduler and also be imported by it can simply implement `Job` without scheduler importing it.

```
// file: app.reporting.ixx
export module app.reporting;

import app.scheduler;

export namespace app {

  export class ReportJob : public Job {
  public:
    void run() override;
  };

}
```

```
// file: app.reporting.cpp
module app.reporting;

namespace app {
  void ReportJob::run() {
    // ...
  }
}
```

10.3.6 Technique 4: Merge the Concept, Split Internals With Partitions

If the cycle is real (the concept is one), make it one module and use partitions to separate the internal structure.

```
// file: app.accounting.ixx
export module app.accounting;

export import :api;

export namespace app::acct {
  // re-exported API goes through :api partition
}
```

```
// file: app.accounting.api.ixx
export module app.accounting:api;

export namespace app::acct {

  export struct LedgerId { unsigned long long v{}; };

  export class Ledger {
  public:
    void post(long long cents);
  private:
    long long balance_{};
  };

}
```

```
// file: app.accounting.impl.ixx
module app.accounting:impl;

namespace app::acct {
```

```
  // internal helpers that used to cause cycles between pseudo-modules
}
```

```
// file: app.accounting.cpp
module app.accounting;

import :api;
import :impl;

namespace app::acct {
  void Ledger::post(long long cents) { balance_ += cents; }
}
```

This turns a graph cycle into a single well-defined boundary.

10.3.7 Technique 5: Isolate Legacy Macro/Config Coupling

If module interfaces depend on macro configuration from legacy headers, cycles appear indirectly. The fix is to define a **stable configuration API** instead of importing macro-state.

```
// file: app.config.ixx
export module app.config;

export namespace app {

  export struct BuildConfig {
    bool enable_tracing{};
    bool enable_fast_path{};
  };

  export const BuildConfig& config();

}
```

```
// file: app.config.cpp
module app.config;

namespace app {
  static BuildConfig g_cfg{ true, true };
  const BuildConfig& config() { return g_cfg; }
}
```

Consumers import `app.config` and branch on values, instead of relying on macro definitions crossing boundaries.

10.3.8 A Repeatable "Cycle Elimination" Workflow

When you encounter a cycle, do this every time:

1. **Name the cycle:** A $\leftrightarrow$ B, and list exactly which exported declarations force imports.
2. **Classify it:** type cycle, service cycle, convenience cycle, config/macro cycle.
3. **Choose one fix:**
   - type cycle $\rightarrow$ opaque handles / PImpl / move shared types,
   - service cycle $\rightarrow$ inversion of control / interfaces / callbacks,
   - convenience cycle $\rightarrow$ split helpers into internal partitions,
   - config/macro cycle $\rightarrow$ stable config API and isolate legacy headers.
4. **Move code, not just declarations:** make sure the import direction reflects *who owns the concept*.
5. **Lock it in:** after breaking the cycle, keep a rule that forbids reintroducing it (review checklist).

10.3.9 Worked Example: Turning a Two-Way Dependency Into a DAG

Suppose you have:

- `app.ui` imports `app.services` to call actions,
- `app.services` imports `app.ui` for progress callbacks.

This is a service cycle disguised as “just callbacks”.
Fix: extract a `app.progress` protocol module that both depend on.

```
// file: app.progress.ixx
export module app.progress;

export namespace app {

  export class Progress {
  public:
    virtual ~Progress() = default;
    virtual void on_start(const char* name) = 0;
    virtual void on_step(int current, int total) = 0;
    virtual void on_done() = 0;
  };

}
```

```
// file: app.services.ixx
export module app.services;

import app.progress;

export namespace app {

  export class Services {
```

```
  public:
    void run_long_task(Progress& p);
  };

}
```

```
// file: app.services.cpp
module app.services;

namespace app {
  void Services::run_long_task(Progress& p) {
    p.on_start("task");
    for (int i = 1; i <= 10; ++i) {
      p.on_step(i, 10);
    }
    p.on_done();
  }
}
```

```
// file: app.ui.ixx
export module app.ui;

import app.services;
import app.progress;

export namespace app {

  export class ConsoleProgress : public Progress {
  public:
    void on_start(const char* name) override;
    void on_step(int current, int total) override;
    void on_done() override;
  };
```

```
  export void run_ui();

}
```

```
// file: app.ui.cpp
module app.ui;

namespace app {

  void ConsoleProgress::on_start(const char* name) { (void)name; }
  void ConsoleProgress::on_step(int current, int total) { (void)current; (void)total; }
  void ConsoleProgress::on_done() {}

  void run_ui() {
    Services s;
    ConsoleProgress p;
    s.run_long_task(p);
  }

}
```

Result: `app.ui` depends on `app.services` and `app.progress`. `app.services` depends only on `app.progress`. No cycles.

10.4 Chapter Summary: What You Should Have After This Phase

At the end of architectural discovery + boundary design + cycle elimination, you should have:

- a **small set of named modules** with stable exported interfaces,
- **internal partitions** for heavy/private code,

- a clear **dependency DAG** (domain at the bottom, UI at the top),
- and a practical rule: **import direction equals ownership direction**.

In the next stage of the rebuild, you scale this structure across the remaining subsystems by repeating the same workflow: discover $\rightarrow$ define contract $\rightarrow$ isolate internals $\rightarrow$ remove cycles $\rightarrow$ lock boundaries.

Implementation & Migration

11.1 Converting Headers

11.1.1 The Real Goal: Convert *Contracts*, Not Files

A successful migration does not start by renaming `.hpp` to `.ixx`. It starts by identifying:

- what is **intended API** (stable, supported, documented behavior),
- what is **accidental exposure** (helpers, macros, private types leaking through headers),
- what is **preprocessor-coupled** (config headers, include-order rules),
- and what is **binary boundary sensitive** (DLL/shared library ABI surface).

Modules make accidental exposure harder, but they also make preprocessor-coupling and cyclic design mistakes impossible to ignore. So the first conversion step is always: **shrink and purify the header** before you turn it into a module interface.

11.1.2 Header Conversion Modes (Choose Per Header)

In a real application you typically mix these approaches during migration:

- **Named module interface unit:** best for stable, widely used APIs with clean dependencies.

- **Header unit (transitional):** useful when you need faster builds or isolation but cannot yet remove macros or heavy includes.
- **Keep as classic header:** for macro-heavy platform glue, third-party headers, and code relying on textual inclusion behavior.

A practical rule:

- If the file *must* leak macros, do not export it as a named module API.
- If the file is a stable contract, prefer a named module interface and keep macros outside.

11.1.3 Converting a Typical Header into a Named Module Interface

Assume you have a legacy header `math_api.hpp`:

```
// math_api.hpp (legacy)
#pragma once
#include <cstdint>

namespace app::math {
  std::int64_t add(std::int64_t a, std::int64_t b);
}
```

A direct module conversion is straightforward:

```
// app.math.ixx (module interface unit)
export module app.math;

import <cstdint>;

export namespace app::math {
  std::int64_t add(std::int64_t a, std::int64_t b);
}
```

```
// app.math.cpp (module implementation unit)
module app.math;

namespace app::math {
  std::int64_t add(std::int64_t a, std::int64_t b) {
    return a + b;
  }
}
```

This is the ideal case: no macros, no include-order coupling, and the contract is minimal.

11.1.4 Global Module Fragment: Keep Preprocessor and Legacy Includes Contained

Sometimes the interface must include legacy headers for declarations, platform types, or third-party integration. When you must use textual includes, isolate them *before* the module declaration:

```
// app.platform.ixx
module;

#ifdef _WIN32
  #include <windows.h>
#else
  #include <unistd.h>
#endif

export module app.platform;

export namespace app::platform {
  using native_handle = void*;
  export native_handle current_process();
}
```

Why this matters:

- includes in the global module fragment do not become part of the module interface *as textual export* (they remain a local implementation technique),
- you minimize macro and include spillover into importers,
- and you keep the exported surface stable while refactoring the internals later.

11.1.5 Preprocessor Pitfalls: What Commonly Breaks During Conversion

The following patterns are frequent migration blockers:

- **Config macros shaping the API:** exporting different declarations based on macros means different importers may see different APIs, which is not a stable contract.
- **Headers that intentionally inject names:** "convenience" headers that `using namespace` or define inline variables, operators, or overload sets globally.
- **ODR-by-include patterns:** inline variable definitions or template specializations relying on include guards rather than a clean single-definition strategy.
- **Macro-based APIs:** logging macros, assert macros, reflection-like macros that expand into declarations.

The fix is to redesign:

- convert API-shaping macros into **real types** (options structs, policies),
- move macro usage into **implementation units** or **non-exported partitions**,
- export stable **functions/classes** that remain identical regardless of build flags.

11.1.6 Header Units as a Transitional Step

Header units can be valuable when:

- you want faster incremental builds without fully redesigning the header yet,
- you need to contain include spam while you carve proper module boundaries,
- you are migrating third-party headers where you do not control macro design.

Do not confuse this with “finished modularization”. It is a bridge technique.

11.2 Creating Module Partitions

11.2.1 Why Partitions Exist

Large module interfaces become unmaintainable if everything is in one file. Partitions allow you to:

- keep the exported surface readable,
- move heavy implementation helpers out of the primary interface,
- reduce rebuild impact when internal code changes,
- prevent accidental export of internal details.

11.2.2 Partition Types

In practice, you use two categories:

- **Exported interface partitions:** part of the public contract, structured into logical sections.
- **Internal partitions:** non-exported implementation detail units, invisible to importers.

A critical rule for correct partition design:

- The **primary module interface** is responsible for exporting the module's interface partitions.

11.2.3 Example: Primary Interface + Exported Partitions + Internal Partitions

```
// app.net.ixx (primary module interface)
export module app.net;

export import :types;
export import :api;

// internal partitions are not exported
import :tls_impl;
import :socket_impl;
```

```
// app.net.types.ixx (exported interface partition)
export module app.net:types;

import <cstdint>;

export namespace app::net {

  export struct Endpoint {
    const char* host{};
    std::uint16_t port{};
  };

  export enum class Protocol {
    Tcp,
    Udp
```

```
  };

}
```

```
// app.net.api.ixx (exported interface partition)
export module app.net:api;

import app.net:types;

export namespace app::net {

  export class Client {
  public:
    Client();
    ~Client();

    bool connect(Endpoint ep, Protocol proto);
    int  send_bytes(const void* data, unsigned long long size);

  private:
    struct Impl;
    Impl* p_; // simplified; use smart pointers in production
  };

}
```

```
// app.net.tls_impl.ixx (internal partition)
module app.net:tls_impl;

namespace app::net {
  // TLS helpers, certificate parsing, etc.
}
```

```
// app.net.socket_impl.ixx (internal partition)
```

```
module app.net:socket_impl;

namespace app::net {
  // OS socket glue, platform headers, etc.
}
```

```
// app.net.cpp (module implementation unit)
module app.net;

import app.net:api;
import :tls_impl;
import :socket_impl;

namespace app::net {

  struct Client::Impl {
    int dummy{};
  };

  Client::Client() : p_(new Impl{}) {}
  Client::~Client() { delete p_; }

  bool Client::connect(Endpoint ep, Protocol proto) {
    (void)ep; (void)proto;
    return true;
  }

  int Client::send_bytes(const void* data, unsigned long long size) {
    (void)data;
    return static_cast<int>(size);
  }

}
```

This layout produces a stable public surface while allowing deep internal refactoring without expanding imports.

11.2.4 Partition Naming and Directory Strategy

A scalable convention:

- Primary: `app.net.ixx`
- Exported partitions: `app.net.types.ixx`, `app.net.api.ixx`
- Internal partitions: `app.net.impl_*.ixx`
- Implementation unit(s): `app.net.cpp`, plus additional implementation units if needed

Avoid scattering partitions randomly. Treat them as a cohesive subsystem.

11.3 Updating Build Systems

11.3.1 What Changes in the Build Model

Headers are text-included; build systems historically guessed dependencies by scanning `#include` lines. Named modules introduce `import` dependencies that require:

- building module interface units **before** importers,
- generating and locating **BMI artifacts** (compiler-specific),
- and accurately tracking module import graphs, including partitions.

Therefore the build system must handle:

- **dependency scanning** for module imports,

- **dynamic ordering** of compilation steps,
- correct propagation of **BMI search paths** to importers.

11.3.2 CMake: Declaring Module File Sets (Modern Pattern)

When using modern CMake module support, treat module interface sources as a dedicated file set.

```
cmake_minimum_required(VERSION 3.28)
project(App LANGUAGES CXX)

add_library(app_core)

target_compile_features(app_core PUBLIC cxx_std_20)

target_sources(app_core
  PRIVATE
    src/app.net.cpp
    src/app.math.cpp
  FILE_SET CXX_MODULES
  BASE_DIRS
    ${CMAKE_CURRENT_SOURCE_DIR}/src
  FILES
    src/app.net.ixx
    src/app.net.types.ixx
    src/app.net.api.ixx
    src/app.math.ixx
)

add_executable(app_main src/main.cpp)
target_link_libraries(app_main PRIVATE app_core)
```

Key points:

- The build system must know which sources are module interfaces/partitions.
- The build must be able to scan `import` lines to create the correct order.
- The generator and toolchain must cooperate to produce BMI artifacts and use them reliably.

11.3.3 Multi-Directory Projects: Controlling Visibility Between Targets

In large projects you will have multiple libraries and executables. A migration-safe rule:

- Each library target owns its module interfaces.
- Executables import library modules *through linking to the library target*, not by manually wiring BMI paths.

This keeps module usage aligned with normal dependency management.

11.3.4 Toolchain Reality: BMI Artifacts Are Not a Portable Interface

A module import relies on compiler-generated artifacts. Across different compilers (and sometimes different versions of the same compiler), these artifacts are not interchangeable. Therefore:

- **build** module interfaces as part of the consumer build (or provide a toolchain-matched package),
- treat module artifacts like **object files** rather than like **headers**,
- be extremely deliberate when designing installation and consumption workflows for modular libraries.

This is why many migrations keep a hybrid packaging strategy during transition.

11.3.5 Incremental Builds and Scanning Cost

Modules can massively reduce compile time from repeated parsing of headers, but they also introduce:

- scanning steps,
- BMI generation,
- and recompile cascades when exported surfaces change.

A practical performance rule:

- Keep exported interfaces small and stable.
- Push heavy code to internal partitions or implementation units.
- Avoid exporting templates unless they truly are part of the stable API story.

11.3.6 A Minimal "Migration Build Matrix" Checklist

During migration, continuously validate these combinations (as applicable to your environment):

- Debug vs Release
- Static library vs Shared library
- Unity build off (recommended during migration) vs on
- Sanitizers (where applicable)

Modules magnify build configuration differences, so you want mismatches to fail early.

11.4 Fixing ABI and Visibility Issues

11.4.1 Critical Clarification: `export` Is Not a Linker Visibility Attribute

In C++, `export module` and `export` on declarations control what is visible to **importers at the language level**. They do *not* automatically solve:

- symbol export/import for shared libraries,
- ABI stability across compiler settings,
- cross-DSO boundaries for templates and inline functions.

You must still design the binary surface intentionally.

11.4.2 Visibility on Shared Libraries: Keep It Explicit

On ELF-based platforms, a common best practice is:

- compile with hidden visibility by default,
- explicitly mark the API that must be exported.

On Windows, you typically:

- use `__declspec(dllexport)` when building the DLL,
- and `__declspec(dllimport)` when consuming it (or rely on modern linker behavior when appropriate).

A portable pattern is to centralize visibility macros, but keep them **out of module contracts when possible**. Prefer exporting stable functions/classes from the module, and apply platform visibility attributes to those declarations in a controlled way.

11.4.3 Avoid Exporting ABI-Fragile Types Across Binary Boundaries

If a module is used across shared library boundaries (plugin architectures, system integrations), do not export:

- STL containers in the public ABI surface,
- types whose layout depends on compiler/library configuration,
- or templates that cause codegen differences across DSOs.

Prefer:

- opaque handles (PImpl),
- C-like stable ABI functions,
- or a narrow abstract interface with ownership clearly documented.

11.4.4 Example: ABI-Stable Module API Using PImpl + Stable Value Types

```
// app.image.ixx
export module app.image;

import <cstdint>;

export namespace app::image {

  export struct Size {
    std::int32_t w{};
    std::int32_t h{};
  };

  export class Decoder {
```

```
  public:
    Decoder();
    ~Decoder();

    bool open_file(const char* path);
    Size size() const;

  private:
    struct Impl;
    Impl* p_;
  };

}
```

```
// app.image.cpp
module app.image;

namespace app::image {

  struct Decoder::Impl {
    Size s{0, 0};
  };

  Decoder::Decoder() : p_(new Impl{}) {}
  Decoder::~Decoder() { delete p_; }

  bool Decoder::open_file(const char* path) {
    (void)path;
    p_->s = Size{1920, 1080};
    return true;
  }

  Size Decoder::size() const {
    return p_->s;
```

```
  }

}
```

The ABI surface is small, stable, and does not leak internal containers or third-party types.

11.4.5 Template and Inline Code: Control Code Generation

Exported templates and inline functions can be correct and useful, but they have cost:

- Exported templates increase BMI complexity and increase the chance that small changes trigger rebuild cascades.
- Inline functions may generate code in multiple TUs, and changes can create larger relink scopes.

Mitigations:

- For heavy templates, consider moving implementation into internal partitions and providing explicit instantiations when the type set is finite and controlled.
- Keep inline bodies small and stable; move logic to non-inline functions in implementation units when possible.

11.4.6 ODR and “Same Definition” Failures During Migration

Even with modules, you can still encounter One Definition Rule violations through:

- duplicated non-inline function definitions in multiple module implementation units,
- multiple conflicting declarations caused by mixing legacy headers and module exports,
- macro-conditioned declarations that create divergent signatures across TUs.

A robust migration rule:

- Any entity that is exported must have a single authoritative declaration in the module interface, and all legacy headers must either be removed or made strictly forwarding-only.

11.4.7 Mixed Worlds: Headers + Modules and "Two Sources of Truth"

During migration, the worst class of bugs is when a type/function is declared in both:

- a legacy header that some TUs include,
- and a module interface that other TUs import.

Symptoms include:

- subtle mismatch in inline bodies,
- different macro-conditioned signatures,
- link-time multiple-definition or unresolved-symbol errors.

Fix strategy:

- Make the legacy header a thin forwarding shell that **imports** the module only where supported, or remove it entirely once all consumers import the module.
- Do not let both remain authoritative for longer than the shortest practical transition window.

11.4.8 Diagnosing Visibility Problems: A Focused Checklist

When a modular shared library fails to link or exports the wrong set of symbols, check:

- Are the exported declarations compiled into the library target, or only into consumers?

- Is default visibility hidden, and did you forget to mark the intended exported ABI surface?
- Are you exporting inline/template-only APIs that do not produce linkable symbols?
- Are you accidentally building two variants of the module (different defines, different stdlib mode)?

11.4.9 Final Migration Discipline: Lock the Contract and Guard It

Once a module is declared stable:

- treat its interface as **public API**,
- review every change to `export` declarations,
- keep a **dependency whitelist** per module,
- and require that any new dependency is justified as contract-level need, not convenience.

11.5 Chapter Summary

By the end of this chapter you should have:

- a repeatable method to convert headers into named modules or transitional header units,
- a partition strategy that keeps interfaces small and internals heavy,
- a build system updated to correctly scan and order module builds,
- and an explicit ABI/visibility policy that keeps modular libraries reliable in real deployments.

Integration, Testing & Performance

12.1 Testing Module Boundaries

12.1.1 Why Module Boundary Testing Is Different

Traditional header-based testing verifies behavior. Module-based testing must additionally verify:

- correctness of exported interfaces,
- absence of accidental transitive exposure,
- stability of dependency direction,
- and isolation of implementation details.

A module boundary is a *contract*. Therefore, it must be tested as a contract, not merely as a collection of functions.

12.1.2 Contract-Level Testing Strategy

Testing module boundaries involves three complementary levels:

1. **Importer-level compilation tests**
2. **Behavioral tests through public API**

3. **Negative compilation tests (boundary enforcement)**

Importer-Level Compilation Tests

Create minimal importer translation units that only:

- import the module,
- use exported declarations,
- do not include any legacy headers.

Example:

```
// test_import_math.cpp
import app.math;

int main() {
  return app::math::add(2, 3) == 5 ? 0 : 1;
}
```

If this fails to compile without including unrelated headers, your module leaks implicit dependencies.

Behavioral Tests Through Public API

Use a test framework or a minimal custom harness. Only interact with exported entities.

```
// test_net.cpp
import app.net;
import <cassert>;

int main() {
  app::net::Endpoint ep{"localhost", 8080};
```

```
  app::net::Client client;
  bool ok = client.connect(ep, app::net::Protocol::Tcp);
  assert(ok);
}
```

The test must not:

- access internal partitions,
- rely on implementation headers,
- depend on macro definitions.

Negative Boundary Tests

You can deliberately test that internals are not visible.

```
// test_internal_visibility.cpp
import app.net;

// This must fail if Impl is not exported:
// app::net::Client::Impl x;  // should be inaccessible
```

Such tests ensure:

- implementation detail types are not exported,
- accidental exposure does not regress.

12.1.3 Dependency Direction Verification

A large modular system must remain acyclic. To enforce this:

- maintain a module dependency diagram,

- reject any new import that violates layering rules,
- review every `import` in code review.

A simple static rule:

- Domain modules import only standard library and utility modules.
- Services import domain.
- UI imports services and domain.
- Infrastructure never imports UI.

Automated checks can be done via:

- scanning import statements,
- build-system dependency graph inspection.

12.1.4 Testing Across Mixed Worlds (Headers + Modules)

During migration, some units include headers while others import modules. You must test:

- both inclusion and import paths do not produce divergent behavior,
- ODR consistency between legacy headers and modules,
- consistent macro configuration across both worlds.

Temporary forwarding headers should be validated carefully.

12.2 Benchmarking Module Builds

12.2.1 Build Performance Model

Module builds differ from header-based builds:

- Module interface units are compiled once into BMI artifacts.
- Importers reuse the compiled module interface.
- Parsing cost shifts from repeated textual inclusion to single interface compilation.

However:

- Changing exported interfaces invalidates all importers.
- Large exported templates increase BMI size and compile cost.

12.2.2 Measurement Baseline Strategy

Before migration:

- measure full clean build time,
- measure incremental build after editing a header,
- measure incremental build after editing an implementation file.

After modularization:

- measure the same three scenarios,
- compare rebuild scope,
- compare total compilation units touched.

12.2.3 Practical Build Benchmark Example

Example baseline scenario:

- 200 translation units,
- heavy shared header included in 150 files.

Legacy change in shared header:

- triggers recompilation of 150 units.

Modular scenario:

- shared header becomes `app.shared` module,
- change in implementation unit only rebuilds module implementation,
- change in exported interface rebuilds module + importers.

The optimization goal:

- keep exported interface stable,
- move volatile code into internal partitions.

12.2.4 Avoiding Performance Regressions

Common mistakes that increase compile time:

- exporting large template libraries unnecessarily,
- exporting inline-heavy algorithm implementations,
- merging too many unrelated APIs into one module.

Optimization strategies:

- split modules by volatility,
- isolate template implementation in partitions,
- use explicit instantiation where appropriate.

12.2.5 Benchmarking Runtime Performance

Modules should not affect runtime semantics. However, potential indirect effects include:

- different inlining decisions,
- altered visibility attributes,
- modified symbol resolution behavior in shared builds.

Benchmark both:

- static builds,
- shared library builds.

Use controlled microbenchmarks:

```
import app.math;
import <chrono>;
import <iostream>;

int main() {
  constexpr int N = 100000000;
  auto start = std::chrono::high_resolution_clock::now();
  long long sum = 0;
```

```
  for (int i = 0; i < N; ++i)
    sum += app::math::add(i, i);
  auto end = std::chrono::high_resolution_clock::now();
  std::cout << (end - start).count() << "\n";
}
```

Ensure identical optimization flags before comparing results.

12.3 Debugging Module Issues

12.3.1 Common Categories of Module Errors

- Missing BMI artifacts
- Incorrect module build order
- Conflicting module names
- Duplicate definitions across partitions
- Macro leakage from global module fragment

12.3.2 Import Failures

Typical message patterns:

- module not found,
- BMI not reachable,
- inconsistent module interface.

Diagnosis steps:

1. Verify module interface compiled successfully.
2. Ensure build system passes correct BMI search paths.
3. Confirm module name matches declaration exactly.
4. Confirm consistent compiler flags across interface and importer.

12.3.3 ODR Violations

Symptoms:

- duplicate symbol at link time,
- inconsistent inline function definitions.

Common causes:

- mixing header includes and module imports,
- redefining non-inline functions in multiple implementation units,
- exporting definitions unintentionally in multiple partitions.

Fix approach:

- centralize exported declarations,
- remove legacy includes after migration,
- review partitions for duplicated definitions.

12.3.4 Debugging Macro Interactions

If behavior differs between builds:

- inspect global module fragment,
- search for macros that affect exported declarations,
- eliminate API-shaping macros.

12.3.5 Symbol Visibility Debugging

In shared library builds:

- inspect exported symbols,
- confirm visibility attributes applied correctly,
- verify consistent build flags between modules and consumers.

12.4 Deploying Module-Based Applications

12.4.1 Build Artifact Strategy

Deployment does not ship BMI artifacts. Deployment ships:

- object files,
- static libraries,
- shared libraries,
- executables.

BMI files are compilation artifacts, not runtime dependencies.

12.4.2 Packaging Modular Libraries

When distributing a modular library:

- provide compiled library binaries,
- provide module interface source files,
- ensure toolchain compatibility.

Avoid assuming cross-compiler BMI portability.

12.4.3 Versioning Policy

Changing exported module interfaces is equivalent to:

- modifying a public header in legacy systems,
- possibly breaking ABI,
- triggering rebuild of importers.

Adopt semantic versioning discipline:

- incompatible exported changes require major version increments,
- additive exports are minor version changes.

12.4.4 Continuous Integration Integration

CI pipelines should:

- build clean from scratch,

- test incremental rebuild scenarios,
- validate dependency graph constraints,
- verify release-mode optimization builds.

12.4.5 Production Deployment Checklist

Before releasing a module-based application:

- confirm no legacy headers are authoritative for public API,
- confirm exported interfaces are minimal,
- confirm no cyclic module imports exist,
- confirm consistent compiler flags across all modules,
- confirm shared-library symbol visibility correctness.

12.5 Chapter Summary

In this chapter, we established:

- systematic methods for testing module boundaries,
- disciplined benchmarking of compile-time and runtime performance,
- structured debugging techniques for module-related failures,
- and reliable deployment strategies for modular applications.

A successful migration to C++ modules is not complete when the code compiles. It is complete when:

- boundaries are stable,
- builds are predictable,
- performance is measurable,
- and deployment is controlled.

Ending Module Volume

Conclusion

What This Volume Actually Proved

This volume did not treat Modules as a marketing feature, nor as a single syntax upgrade. It treated Modules as an *architectural forcing function* that turns implicit structure into explicit structure.
If you remember one idea from everything you read, it should be this:

- **Modules do not create architecture.**
- **Modules reveal architecture.**

A modular codebase is not defined by `export module`. It is defined by:

- stable boundaries,
- intentionally designed interfaces,
- controlled dependency direction,
- and a build system that can reliably represent the import graph.

This is why real migrations succeed only when they start with discovery and refactoring, not file conversion.

The Core Outcomes You Should Have By Now

By the end of this volume, you should be able to do the following in a real production codebase:

1. **Describe the system as a dependency graph**, not as a directory tree.
2. **Identify stable contracts** and separate them from volatile implementation.
3. **Create named module boundaries** that align with ownership and responsibility.
4. **Design exported interfaces** with minimal transitive exposure.
5. **Eliminate cycles** by redesigning contracts (not by hacking build order).
6. **Migrate incrementally** using a mix of named modules, partitions, and transitional techniques.
7. **Update build pipelines** to correctly scan, order, and cache module artifacts.
8. **Verify correctness** through boundary testing, import tests, negative tests, and integration suites.
9. **Measure performance honestly** (clean builds, incremental builds, and runtime impact).
10. **Ship reliably** with clear ABI policy, symbol visibility discipline, and packaging rules.

If any of these outcomes are missing, the migration is not complete; it is merely compiling.

The Non-Negotiables of a Successful Modules Migration

In practice, nearly every failure case comes from violating one of the following rules.

Non-Negotiable #1: Interface Stability Is the Primary Optimization

Modules improve build performance when exported surfaces are small and stable. If the exported surface changes daily, modules can become an amplifier of rebuild cascades.
A stable module contract should:

- export only what the rest of the program truly needs,
- avoid exporting volatile utilities and helper patterns,
- keep templates deliberate rather than accidental,
- push heavy or frequently changing logic into implementation units or internal partitions.

Non-Negotiable #2: Dependency Direction Must Reflect Ownership

If module A "owns" the concept, module B should depend on A, not the opposite. When that is not true, it signals a design mistake:

- a concept was split incorrectly,
- or an interface leaks responsibilities,
- or a shared vocabulary was not extracted properly.

Non-Negotiable #3: Macros Cannot Be Your API Model

If your "public API" is effectively a macro language that reshapes declarations, you do not have a stable contract. You have multiple contracts disguised as one.
The migration strategy is always the same:

- replace macro-shaped API choices with real types (options, policies, configuration objects),

- isolate platform glue and third-party headers behind implementation boundaries,
- export stable functions and types that remain consistent across build modes.

Non-Negotiable #4: Build System Must Become "Module-Aware"

Modules change the build problem from "compile independent translation units" to "compile an import graph in a correct order".
A module-aware build pipeline must:

- discover module interfaces and partitions,
- scan imports reliably,
- compile interfaces before importers,
- propagate the required artifacts consistently.

If the build system treats modules like headers, you will get fragile builds, random failures, and expensive debugging.

Non-Negotiable #5: ABI and Visibility Are Still Your Responsibility

Language-level export is not the same as binary-level export. A module can define a clean language contract while still producing:

- unstable ABI across toolchains,
- broken symbol visibility in shared libraries,
- accidental cross-DSO template and inline issues.

A professional migration explicitly defines:

- what is allowed to cross the shared-library boundary,
- how symbols are exported and hidden,
- which types are ABI-safe and which must be wrapped or hidden.

What Modules Change, and What They Do Not

What Modules Change

Modules change:

- the compilation model (imports instead of textual includes),
- the visibility model (exported vs internal declarations),
- the scale behavior of large builds (reduced repeated parsing when interfaces are stable),
- the architecture pressure (cycles and accidental coupling become obvious and costly).

What Modules Do Not Change

Modules do not automatically fix:

- poor layering decisions,
- unstable or bloated APIs,
- binary compatibility mistakes,
- unclear ownership and responsibilities,
- testing discipline or deployment hygiene.

Modules reward good architecture and punish hidden architecture.

A Final Practical Blueprint for Real Teams

If you are leading a real migration, this blueprint is a safe operational path:

Phase 1: Discovery and Contract Definition

- Inventory subsystems, identify stable concepts, map dependency direction.
- Create contract sheets for each boundary (types, errors, threading, ownership).
- Identify macro dependencies and include-order coupling.

Phase 2: Boundary Carving and Cycle Elimination

- Convert the smallest stable API first (domain/vocabulary).
- Break cycles by extracting vocabulary modules and applying inversion of control.
- Use internal partitions for heavy implementation glue.

Phase 3: Migration at Scale

- Convert headers that represent true contracts into named module interfaces.
- Keep transitional techniques for legacy zones, but continuously reduce them.
- Ensure no “two sources of truth” exist (do not keep both a legacy header and a module interface authoritative).

Phase 4: Integration, Testing, and Performance Validation

- Add importer-only compilation tests per module.

- Add negative tests to prevent accidental exports.
- Benchmark clean builds and incremental builds using controlled scenarios.
- Validate runtime performance in optimized builds and shared-library configurations.

Phase 5: Deployment Discipline

- Define an ABI policy and symbol visibility policy.
- Package and distribute module-based libraries with toolchain alignment in mind.
- Lock interface changes behind versioning rules and review gates.

Common Traps That Mature Teams Still Fall Into

1. **Exporting too much:** turning a module into a "mega header" with transitive import flood.
2. **Exporting volatility:** exporting helpers and templates that change frequently.
3. **Using modules to avoid design work:** expecting the compiler to solve cycles instead of redesigning contracts.
4. **Confusing language visibility with binary visibility:** assuming `export` solves shared-library exporting.
5. **Keeping dual APIs alive:** allowing both headers and modules to be authoritative for the same entity.
6. **Ignoring build configuration consistency:** mixing flags across module interfaces and importers.

Avoiding these traps is not a matter of talent; it is a matter of discipline.

Looking Forward: What to Expect as Modules Mature Toward C++26

From the perspective of real engineering teams, the most important evolution is not new keywords. It is the ecosystem maturing around modules:

- more reliable build-system scanning and ordering,
- clearer packaging conventions for modular libraries,
- improved diagnostics for module-related failures,
- more predictable integration patterns for large polyglot and multi-target systems.

In other words: the success of modules in production depends as much on tooling, build pipelines, and packaging practice as it does on the core language model.

Final Statement

The end of a modules volume is not the end of the story. It is the moment you stop asking “How do I convert my headers?” and start asking:

- What is my program’s true architecture?
- Which contracts are stable enough to export?
- What dependencies am I willing to support long-term?
- How do I enforce those answers through code review, tests, and builds?

If you treat Modules as architecture, not syntax, you will get:

- clearer boundaries,
- fewer accidental dependencies,
- more predictable builds,
- and a codebase that scales in both size and team ownership.

That is the real promise of Modules for modern C++ engineering.

Appendices

Appendix A: Canonical File Layout Patterns for Large Modular Codebases

This appendix provides practical, repeatable layouts that scale beyond small demos. The goal is to make a large application predictable for:

- humans (discoverability, reviewability),
- build systems (module scanning and ordering),
- and long-term evolution (stable exported surfaces, volatile internals).

A.1 Minimal Single-Module Pattern

Use when the module is small and stable.

```
src/
  app.math.ixx          (module interface)
  app.math.cpp          (module implementation unit)
  main.cpp              (imports app.math)
```

```
// app.math.ixx
export module app.math;
```

```
import <cstdint>;

export namespace app::math {
  std::int64_t add(std::int64_t a, std::int64_t b);
}
```

```
// app.math.cpp
module app.math;

namespace app::math {
  std::int64_t add(std::int64_t a, std::int64_t b) { return a + b; }
}
```

A.2 Scalable Pattern: Primary Interface + Exported Partitions + Internal Partitions

Use when:

- interface must be logically structured,
- internals are heavy or volatile,
- platform glue must be isolated.

```
src/
  app.net.ixx                (primary interface)
  app.net.types.ixx          (exported partition)
  app.net.api.ixx            (exported partition)
  app.net.impl.socket.ixx    (internal partition)
  app.net.impl.tls.ixx       (internal partition)
  app.net.cpp                (module implementation unit)
```

A clean primary interface that re-exports only the contract:

```
// app.net.ixx
export module app.net;

export import :types;
export import :api;

import :impl.socket;
import :impl.tls;
```

Exported partitions:

```
// app.net.types.ixx
export module app.net:types;
import <cstdint>;

export namespace app::net {
  export struct Endpoint { const char* host{}; std::uint16_t port{}; };
  export enum class Protocol { Tcp, Udp };
}
```

```
// app.net.api.ixx
export module app.net:api;
import app.net:types;

export namespace app::net {
  export class Client {
  public:
    Client();
    ~Client();
    bool connect(Endpoint ep, Protocol proto);
  private:
    struct Impl;
    Impl* p_;
  };
}
```

Internal partitions (not exported):

```
// app.net.impl.socket.ixx
module app.net:impl.socket;

namespace app::net {
  // platform socket glue, helpers, etc.
}
```

```
// app.net.impl.tls.ixx
module app.net:impl.tls;

namespace app::net {
  // TLS helpers, parsing, etc.
}
```

Implementation unit:

```
// app.net.cpp
module app.net;

import app.net:api;
import :impl.socket;
import :impl.tls;

namespace app::net {

  struct Client::Impl { int dummy{}; };

  Client::Client() : p_(new Impl{}) {}
  Client::~Client() { delete p_; }

  bool Client::connect(Endpoint ep, Protocol proto) {
    (void)ep; (void)proto;
```

```
    return true;
  }

}
```

A.3 Pattern for “Legacy Header Containment”

When a module must interact with macro-heavy headers, isolate them in the global module fragment.

```
// app.platform.ixx
module;

#ifdef _WIN32
  #include <windows.h>
#else
  #include <unistd.h>
#endif

export module app.platform;

export namespace app::platform {
  export int pid();
}
```

Then keep the implementation free to refactor away the header dependency later.

Appendix B: Conversion Playbook for Header-to-Module Migration

This playbook is designed for large programs where headers evolved into de-facto APIs.

B.1 The 7-Step Conversion Checklist

For each candidate header:

1. **Shrink:** remove accidental exports (helpers, unrelated utilities).
2. **Stabilize:** remove API-shaping macros; replace with real types/options.
3. **Split:** separate contract from implementation (PImpl, internal helpers).
4. **Name:** choose a module name matching architecture ownership (`app.domain`, `app.storage`).
5. **Partition:** move heavy/volatile implementation to internal partitions.
6. **Bridge:** keep legacy headers as forwarding shells only for a short transition window.
7. **Lock:** add boundary tests so the contract cannot regress quietly.

B.2 The "Two Sources of Truth" Anti-Pattern and the Safe Bridge

During migration, the most dangerous situation is when some code includes a header while other code imports a module that defines the same API.
Safe bridge approach (short-lived):

- keep the module interface as the **single authoritative declaration**,
- keep the legacy header as a **thin forwarding facade** that declares as little as possible,
- enforce that new code **imports the module** rather than including the header.

B.3 Converting Macro-Based “Configuration Headers”

Replace:

```
// legacy_config.hpp
#pragma once
#define APP_ENABLE_TRACING 1
#define APP_FAST_PATH 1
```

With an exported configuration API:

```
// app.config.ixx
export module app.config;

export namespace app {
  export struct BuildConfig {
    bool tracing{};
    bool fast_path{};
  };

  export const BuildConfig& config();
}
```

```
// app.config.cpp
module app.config;

namespace app {
  static BuildConfig g{true, true};
  const BuildConfig& config() { return g; }
}
```

This eliminates macro coupling across boundaries and makes configuration explicit.

Appendix C: Partitions Deep Dive

C.1 Partition Design Rules That Prevent Long-Term Pain

- Keep exported partitions small and stable; treat them as public headers.
- Internal partitions may be numerous; they are for refactoring freedom.
- Do not export internal helper namespaces by accident.
- Prefer a single primary interface that re-exports exported partitions (one import for the concept).

C.2 "Volatility Partitioning": Separate Stable API From Fast-Changing Code

A scalable technique is to split code by rate of change:

```
app.analytics.ixx            (exports stable facade)
app.analytics.api.ixx        (exports stable contract)
app.analytics.impl.v1.ixx    (internal, volatile)
app.analytics.impl.exp.ixx   (internal, experimental)
app.analytics.cpp            (ties together)
```

This reduces rebuild cascades when fast-changing pieces evolve.

Appendix D: Build Systems and Toolchain Integration

This appendix focuses on practical build-system work rather than theory.

D.1 CMake: A Modern Pattern Using a Module File Set

A recommended approach is to declare module interface and partition sources explicitly.

```
cmake_minimum_required(VERSION 3.28)
project(App LANGUAGES CXX)

add_library(app_core)

target_compile_features(app_core PUBLIC cxx_std_20)

target_sources(app_core
  PRIVATE
    src/app.net.cpp
    src/app.math.cpp
  FILE_SET CXX_MODULES
    BASE_DIRS ${CMAKE_CURRENT_SOURCE_DIR}/src
    FILES
      src/app.net.ixx
      src/app.net.types.ixx
      src/app.net.api.ixx
      src/app.math.ixx
)

add_executable(app_main src/main.cpp)
target_link_libraries(app_main PRIVATE app_core)
```

Operational notes:

- Keep module interfaces and partitions in the `CXX_MODULES` file set.
- Keep module implementation units as normal `PRIVATE` sources.
- Avoid unity builds during early migration; they can hide ordering and ODR problems.

D.2 MSVC Command-Line Concepts (IFC Artifacts and Search Paths)

MSVC's modules model typically involves:

- compiling interface units to produce both object code and interface artifacts,
- ensuring importers can find interface artifacts through configured search directories,
- keeping compiler flags consistent between interface compilation and importer compilation.

A minimal conceptual example:

```
REM Compile module interface (produces .obj and interface artifact)
cl /std:c++20 /c /interface src\app.math.ixx

REM Compile importer
cl /std:c++20 /c src\main.cpp

REM Link
cl src\app.math.obj src\main.obj
```

In practice, your build system must coordinate output directories and search paths so importers locate artifacts reliably.

D.3 Clang Concepts: Dependency Discovery and Prebuilt Module Paths

For Clang-based toolchains, large builds often rely on:

- dependency discovery (so build order is correct),
- prebuilt module paths (so importers find the built interfaces),
- consistent flags across all compilations.

A conceptual two-step pattern:

- build module interfaces into a known directory,
- compile importers with that directory configured as a prebuilt-module search location.

D.4 GCC Concepts: The Module Cache and Why Clean Builds Matter

GCC uses a module cache mechanism for compiled module interface artifacts. Practical guidance for stable results:

- keep compiler flags consistent across interface and importers,
- do clean builds when changing build flags,
- isolate build directories per configuration (Debug/Release) to avoid cross-contamination.

D.5 A Migration-Friendly CI Build Matrix

During migration, treat modules as a build-system change and test the matrix early:

- Debug + Release
- Static + Shared (if applicable)
- With/without LTO (if used in production)
- With sanitizers (where supported)

The point is not to test everything forever, but to expose configuration-sensitive module issues early.

Appendix E: Diagnostics and Debugging Field Manual

E.1 Failure Modes You Will See in Real Migrations

- **Import fails:** artifact not found, wrong search path, wrong module name.
- **Build order wrong:** importer compiles before interface; partition not built first.
- **Configuration mismatch:** interface compiled with different defines/options than importer.
- **ODR/link errors:** duplicated definitions caused by mixed headers+modules or duplicated partition definitions.
- **Macro leakage:** global module fragment injects macros that change exported declarations unexpectedly.
- **Tooling gaps:** IDE indexers and language servers may lag behind full modules semantics in some workflows.

E.2 A Practical Triage Checklist

When a module import fails or behaves inconsistently:

1. Verify the module interface unit compiles in isolation.
2. Verify the importer is using the same language mode and standard version.
3. Verify the importer sees the built interface artifacts (search paths).
4. Verify the module name exactly matches the `export module` declaration.
5. Verify configuration flags (defines, ABI toggles, runtime selection, visibility flags) match.

6. Verify you have not accidentally introduced “two sources of truth” (header include vs module import).
7. If the error persists, do a clean build of the entire graph.

E.3 “Two Sources of Truth” Link Failure Example

Common symptom:

- One TU includes a header that defines an inline function or variable.
- Another TU imports a module that provides a different definition of the same entity.

Mitigation pattern:

- ensure the module interface is the only authoritative definition,
- make legacy headers either forward-only or removed after transition.

E.4 Visibility vs Language Export: The Mental Model

- **Language export** controls what importers can see at compile time.
- **Binary visibility** controls what linkers and dynamic loaders can resolve at link/run time.

You must handle both. Modules are not a replacement for a shared-library export policy.

Appendix F: ABI and Visibility Templates

F.1 A Cross-Platform Visibility Header (Use Sparingly in Module Interfaces)

In real products you often centralize visibility attributes. Keep this small and stable.

```
// app.visibility.hpp (classic header; usually included in global module fragment if needed)
#pragma once

#if defined(_WIN32)
  #if defined(APP_BUILDING_DLL)
    #define APP_API __declspec(dllexport)
  #else
    #define APP_API __declspec(dllimport)
  #endif
#else
  #define APP_API __attribute__((visibility("default")))
#endif
```

Use in a module interface only if your deployment model requires it:

```
// app.api.ixx
module;
#include "app.visibility.hpp"

export module app.api;

export namespace app {
  export class APP_API Engine {
  public:
    Engine();
    int run();
  };
}
```

F.2 ABI-Safe Surface Pattern

Across shared-library boundaries, prefer:

- small POD-like value types,

- opaque handles,
- stable abstract interfaces (careful with ownership),
- avoiding STL containers in the ABI surface.

Example:

```
// app.image.ixx
export module app.image;
import <cstdint>;

export namespace app::image {

  export struct Size { std::int32_t w{}; std::int32_t h{}; };

  export class Decoder {
  public:
    Decoder();
    ~Decoder();

    bool open_file(const char* path);
    Size size() const;

  private:
    struct Impl;
    Impl* p_;
  };

}
```

Appendix G: Boundary Testing Recipes

G.1 Import-Only Compilation Tests

A minimal importer test ensures the module contract is self-sufficient.

```
// test_import_app_net.cpp
import app.net;
import <cassert>;

int main() {
  app::net::Endpoint ep{"localhost", 8080};
  app::net::Client c;
  bool ok = c.connect(ep, app::net::Protocol::Tcp);
  assert(ok);
}
```

G.2 Negative Compilation Tests

The boundary should prevent accidental internal access. A negative test intentionally tries to use non-exported entities and must fail compilation.

```
// test_negative_visibility.cpp
import app.net;

// Example of forbidden access (must fail if uncommented):
// app::net::Client::Impl x;
int main() {}
```

In CI, you can structure this as a build target expected to fail.

G.3 Dependency Direction Tests (Import Policy Enforcement)

You can enforce architectural rules with a simple policy:

- Domain must not import Infrastructure.
- Infrastructure must not import UI.

A pragmatic enforcement method:

- scan module source files for forbidden imports (simple text scanning is often enough),
- or use build graph inspection if your build system exposes import dependencies.

Appendix H: Build Benchmarking Toolkit

H.1 What to Measure

For every major step in migration, measure:

- **Clean build time** (from an empty build directory)
- **Incremental build time** after editing:
  - an exported interface,
  - an internal partition,
  - an implementation unit,
  - a non-modular header (legacy zone).
- **Rebuild scope** (how many files are recompiled)

H.2 Minimal Timing Scripts

POSIX shell example:

```
#!/usr/bin/env bash
set -euo pipefail

build_dir="${1:-build}"
cmake -S . -B "$build_dir" -DCMAKE_BUILD_TYPE=Release
/usr/bin/time -p cmake --build "$build_dir" -j
```

PowerShell example:

```
Param([string]$BuildDir = "build")
cmake -S . -B $BuildDir -DCMAKE_BUILD_TYPE=Release
Measure-Command { cmake --build $BuildDir --config Release } | Select-Object TotalSeconds
```

H.3 A Microbenchmark Skeleton for Runtime Sanity Checks

```
import <chrono>;
import <iostream>;
import app.math;

int main() {
  constexpr int N = 10000000;
  auto t0 = std::chrono::high_resolution_clock::now();

  long long sum = 0;
  for (int i = 0; i < N; ++i) sum += app::math::add(i, i);

  auto t1 = std::chrono::high_resolution_clock::now();
  std::cout << (t1 - t0).count() << "\n";
  return static_cast<int>(sum == 0);
}
```

Use identical optimization flags for fair comparisons.

Appendix I: Deployment and Packaging Notes for Module-Based Applications

I.1 What You Deploy (And What You Do Not)

You deploy:

- executables,
- shared libraries,
- static libraries (as part of build pipelines),
- runtime assets (configs, resources).

You do not deploy:

- compiler module interface artifacts (they are build-time artifacts, not runtime dependencies).

I.2 Distributing a Modular Library to External Consumers

Because module interface artifacts are toolchain-specific, external distribution typically requires one of these strategies:

- **Source distribution of module interfaces:** consumers build interfaces with their toolchain.
- **Toolchain-locked binary distribution:** you ship binaries and module artifacts for a specific compiler+version.

- **Dual distribution (transition):** keep classic headers as the public interface while modules are used internally.

Choose deliberately based on your ecosystem and support obligations.

I.3 Versioning Rules for Exported Interfaces

Treat exported module changes like public header changes:

- breaking changes require major version increments,
- additive exports can be minor version increments,
- internal partition changes should not affect consumers (by design).

I.4 Production Readiness Checklist

Before shipping a module-based application:

- No cyclic module imports remain.
- Exported surfaces are small, stable, and reviewed.
- No macro-based API shaping remains in exported contracts.
- Build flags are consistent across module interfaces and importers.
- Shared library symbol visibility is verified (if you ship shared libraries).
- CI tests include import-only tests and a clean build from scratch.

Appendix J: Quick Reference Tables

J.1 When to Use Each Technique

Technique	Use When
Named module interface	The API is stable and should be imported widely
Exported partitions	The public contract is large but needs structure
Internal partitions	Implementation is heavy/volatile or platform-specific
Global module fragment	You must include macro-heavy legacy headers safely
PImpl in exported API	You need ABI stability or want to hide heavy deps
Header units (bridge)	You need transitional wins but cannot redesign yet
Facade module	You want controlled re-exports and a simple import experience

J.2 Symptoms and Likely Causes

Symptom	Likely Cause
Import fails	Interface artifact not built or not found via search path
Random build failures	Incorrect scanning/order or inconsistent flags
Link duplicates	Mixed header include + module import defining same entity
Huge rebuild cascades	Exported interface too large or too volatile
Different behavior by config	Macro leakage or config macros shaping exported API

References

Scope of This References Chapter

This chapter lists primary, authoritative resources used to ground the Modules material in this volume:

- the ISO C++ language standard and working drafts for normative language rules,
- WG21 papers for the build/dependency-scanning ecosystem around modules,
- official compiler documentation for practical implementation, flags, artifacts, and diagnostics,
- official build-system documentation for module-aware builds, installation, and packaging workflows.

A. ISO C++ Standard and Working Drafts

- **ISO/IEC 14882:2020** *Programming Languages — C++ (C++20).*
  - Primary normative definition of the Modules facility (module units, purviews, declarations, linkage and reachability rules).

  - Baseline for named modules, module interface units, module implementation units, and module partitions.

- **ISO/IEC 14882:2024** *Programming Languages — C++ (C++23 edition).*
  - Post-C++20 standard edition consolidating Modules wording and related core language refinements.
  - Used for cross-checking modern wording and stable-name references.

- **ISO/IEC JTC1/SC22/WG21 Working Drafts (C++26 draft series).**
  - Used for tracking incremental wording and direction as the language evolves toward C++26.
  - Treated as draft material, not final normative text.

B. WG21 Papers for Build and Dependency Scanning

- **P1689R5** *Format for describing dependencies of source files* (WG21 paper).
  - Dependency-discovery interchange format designed to communicate resources required to compile a translation unit, including named modules and header units.
  - Forms a conceptual bridge between compilers and build systems for correct dynamic build graph construction.

- **ISO/IEC JTC1/SC22/WG21 N4860** *Working Draft, Standard for Programming Language C++* (historical C++20-era draft).
  - Useful to understand C++20-era Modules wording evolution and committee intent around the initial standardization.
  - Included here as historical supporting context for migration narratives and toolchain timelines.

C. Microsoft Visual C++ (MSVC) Official Documentation

- **MSVC compiler option documentation:** `/interface`
  - Declares compilation as a module interface unit and defines how interface artifacts are produced and named.
  - Used to ground command-line and project-system behavior for module interface compilation.
- **MSVC compiler option documentation:** `/ifcOutput`
  - Controls where the compiler writes IFC artifacts (interface metadata files).
  - Essential for deterministic build layouts and for multi-target/multi-config directory hygiene.
- **MSVC compiler option documentation:** `/reference`
  - Enables referencing prebuilt module IFCs (and associated behaviors and constraints).
  - Important for multi-project solutions, external module consumption, and migration staging.
- **Microsoft C++ Team Blog series:** *Using C++ Modules in MSVC from the Command Line* (multi-part).
  - Practical, implementation-oriented guidance on interface artifacts, object outputs, and command-line build workflows.
  - Used to anchor real-world flag usage patterns and build ordering concepts.

D. LLVM/Clang Official Documentation

- **Clang documentation:** *Standard C++ Modules*
  - Describes Clang's support for named modules and header units under the C++ standard model.
  - Includes guidance on dependency discovery, build integration, and practical constraints.
- **LLVM development references:** Clang dependency scanning and P1689-oriented workflows
  - Used for understanding how tooling can emit dependency information suitable for build systems.
  - Serves as background for diagnosing scanning and build-graph issues in modern modular builds.

E. GCC Official Documentation

- **GCC Online Documentation (current release manuals).**
  - Authoritative reference for GCC driver behavior, compilation stages, and option semantics.
  - Used for verifying toolchain assumptions and for grounding build-system notes when targeting GCC-based environments.
- **GCC manual pages and option references (system documentation where applicable).**
  - Useful for operational details in scripted builds and CI pipelines.
  - Complements the online manuals for distribution environments.

F. CMake Official Documentation and Kitware Guidance

- **CMake documentation:** `target_sources()` with `FILE_SET` type `CXX_MODULES`
  - Defines how module interface and partition units are declared to CMake so that IDE integration and module-aware behavior can be applied.
  - Used to ground canonical build snippets in this volume.
- **CMake target properties:** `CXX_MODULE_SETS`, `CXX_MODULE_SET`
  - Defines how module file sets are represented and queried at the target level.
  - Used to explain how large multi-target projects should expose and propagate module sources intentionally.
- **CMake installation rules:** `install()` behavior for PUBLIC module file sets
  - Relevant to packaging, distribution, and consumer builds that must compile module interfaces with a matching toolchain.
- **Kitware guidance:** *import CMake; C++20 Modules*
  - Provides build-system perspective on dynamic dependency discovery and the practical realities behind module-aware build graphs.
  - Used to motivate why module support in build tools cannot be treated like header scanning.

G. Notes on Reading and Applying These References

- **Normative vs operational truth:** the ISO standard defines the language rules, while compiler and build-system docs define the operational toolchain behaviors you must satisfy in practice.

- **Drafts vs published standards:** drafts are valuable for direction and intent; published ISO editions remain the stable baseline for correctness claims.
- **Artifacts are toolchain-specific:** module interface artifacts (IFC/BMI equivalents) are build-time products that depend on the compiler and often the exact compiler version; treat them like object files, not like headers.