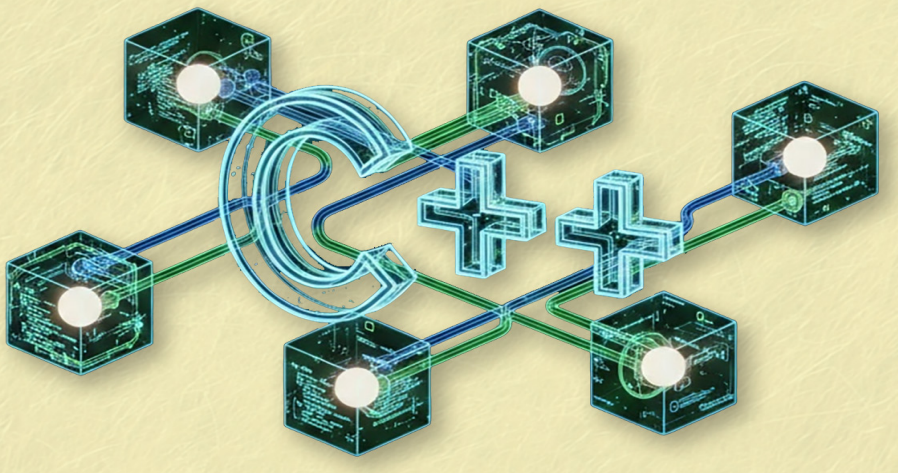
https://simplifycpp.org

# C++ Module

## A Concise Version for Practical Use



Prepared by Ayman alheraki

# C++ Modules

A Concise Version for Practical Use

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

# Contents

# Author's Introduction

This booklet was written for professional C++ developers working on medium to large codebases—systems that are expected to live for years, sometimes decades, and to evolve continuously under real-world constraints.

In such systems, the primary challenges are rarely algorithmic. They are architectural. They arise from uncontrolled dependencies, unclear boundaries, slow and unpredictable build times, and code that becomes increasingly difficult to reason about as it grows.

For decades, `#include` has been the foundation of C++ program composition. This mechanism, inherited from the C language, is fundamentally textual. It performs no semantic validation, enforces no boundaries, and provides no notion of interface versus implementation. Every included header becomes part of the translation unit, whether it is needed or not.

While this model works adequately for small programs, it scales poorly. As systems grow:

- Compilation times increase dramatically
- Seemingly unrelated changes trigger widespread rebuilds
- Dependencies become implicit and difficult to trace
- Internal details leak across module boundaries
- The One Definition Rule becomes fragile and error-prone

Over time, these issues stop being inconveniences and start becoming structural risks. They slow development, discourage refactoring, and make long-term maintenance costly and uncertain.

C++ Modules introduce a fundamentally different compilation model. Instead of textual inclusion, they rely on semantic import. Interfaces are compiled once, validated by the compiler, and consumed as well-defined units. Only explicitly exported declarations are visible. Implementation details remain private by default.

This shift enables:

- Clear and enforceable architectural boundaries
- Stronger encapsulation at the language level
- Predictable and scalable build behavior
- Reduced coupling between components

However, modules are often misunderstood. Some developers view them as a simple replacement for headers. Others attempt to adopt them all at once, expecting immediate results, only to encounter tooling or build system friction.

This booklet avoids both extremes.

Its goal is not to promote modules as a trend, nor to present them as a theoretical language feature. Instead, it treats modules as an engineering tool—one that must be understood, applied carefully, and integrated incrementally.

This work focuses on practical usage. It explains what modules are at a conceptual level, how they actually work in modern compilers, and how they interact with existing C++ codebases. Special attention is given to realistic migration strategies that allow teams to adopt modules gradually, without disrupting production systems or rewriting large amounts of stable code.

If you have ever hesitated to refactor because of build costs, struggled with hidden dependencies, or questioned whether your architecture could survive another decade of growth, this booklet was written for you.

# Preface

C++ has always been a language designed around performance, determinism, and direct control over system resources. From its earliest days, it has been used to build operating systems, compilers, databases, game engines, and large-scale infrastructure where efficiency and predictability are non-negotiable.

However, while the language itself has evolved dramatically, its compilation model remained largely unchanged for decades. That model was inherited from the C language and was never designed to support the scale, complexity, and longevity of modern C++ software systems.

Header files encourage implicit coupling. A single `#include` silently pulls large portions of a codebase into a translation unit, often far beyond what is actually required. Macros ignore scope and visibility rules, leaking across logical boundaries and making code harder to reason about. The One Definition Rule, while essential, becomes increasingly difficult to enforce as the number of headers, libraries, and build configurations grows.

As projects scale, these problems compound. Build systems become intricate and fragile. Small changes trigger widespread recompilation. Developers begin to structure code around build limitations rather than architectural clarity. Over time, the compilation model itself becomes a limiting factor in how systems can evolve.

C++ Modules address these issues by introducing a fundamentally stronger foundation for program composition. They replace textual inclusion with a semantic model that allows the compiler to understand interfaces, dependencies, and visibility explicitly.

By doing so, modules introduce:

- Explicit and well-defined module interfaces
- A clear and enforceable separation between interface and implementation
- Stronger, compiler-enforced guarantees around the One Definition Rule
- Faster, more predictable, and more scalable build behavior

These changes are not merely incremental improvements. They enable a different way of thinking about C++ architecture—one where boundaries are intentional, dependencies are visible, and compilation costs scale linearly rather than exponentially.

This booklet presents C++ Modules not as a theoretical language feature or an academic experiment, but as a practical architectural tool. Its purpose is to show how modules can be used today to build long-lived, maintainable C++ systems that remain robust as they grow in size, complexity, and lifespan.

# Chapter 1

# Why C++ Modules Exist

## 1.1 The Limitations of Header Files

The traditional `#include` mechanism is the historical foundation of C++ program composition. Despite its widespread use, it is important to understand that `#include` is not a language-level composition feature. It is a purely textual substitution performed by the preprocessor before the compiler ever sees the program.
When a header is included, its contents are copied verbatim into every translation unit that includes it. The compiler then parses the same declarations repeatedly, once per translation unit, with no awareness that the same information has already been processed elsewhere.

### 1.1.1 Repeated Parsing and Compile-Time Cost

Consider the following header:

```
// math_utils.h
#pragma once
```

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

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

Now assume this header is included in many source files:

```cpp
// file1.cpp
#include "math_utils.h"

// file2.cpp
#include "math_utils.h"

// ...
```

Each translation unit independently parses the contents of `math_utils.h` and all of its transitive includes. Even if the header never changes, the compiler repeats this work for every translation unit. In large systems with hundreds or thousands of source files, this leads to significant and unnecessary compile-time overhead.

### 1.1.2 Fragile Dependency Graphs

Headers encourage implicit dependencies. A source file may rely on declarations that arrive indirectly through other headers, without explicitly stating that dependency.

```cpp
// a.h
#pragma once
#include "b.h"

void func_a();

// b.h
#pragma once
```

```
#include <string>

void func_b(const std::string&);

// main.cpp
#include "a.h"

int main() {
    std::string s = "test";
    func_b(s);
}
```

Here, `main.cpp` compiles only because `a.h` happens to include `b.h`, which happens to include `<string>`. The dependency on `<string>` is implicit and fragile. If the include structure changes, unrelated files may fail to compile.

### 1.1.3 Accidental Symbol Exposure

Headers expose everything they contain. There is no language-level mechanism to distinguish public API from internal implementation details.

```
// logger.h
#pragma once

void log_info(const char*);

namespace detail {
    void open_log_file();
    void close_log_file();
}
```

Although the functions inside `detail` are intended to be internal, any file including `logger.h` can call them. Over time, such internal details often become part of the de facto

public API, making future refactoring risky.

### 1.1.4 One Definition Rule Pitfalls

Headers frequently contribute to One Definition Rule (ODR) violations, especially when they contain variables, inline functions, or templates.

```
// config.h
#pragma once

const int buffer_size = 1024;
```

When included in multiple translation units, this header produces multiple definitions of `buffer_size`, resulting in linker errors or undefined behavior. As codebases grow, diagnosing and preventing such issues becomes increasingly difficult.

### 1.1.5 Summary of Header-Based Limitations

As projects scale, the header-based model leads to:

- Long and unpredictable compile times
- Implicit and fragile dependency chains
- Poor encapsulation and unintended API exposure
- Increased risk of One Definition Rule violations
- Architectures constrained by build mechanics

These are not incidental flaws. They are structural limitations of the model itself.

## 1.2 The Need for a Semantic Model

C++ Modules were introduced to address these limitations at the language level. Instead of relying on textual substitution, modules introduce a semantic compilation model in which interfaces, dependencies, and visibility are explicit and compiler-enforced.

### 1.2.1 From Textual Inclusion to Semantic Import

With modules, a unit explicitly declares what it exports and what it imports.

```
export module math.core;

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

This module interface is compiled once into a binary module interface. Consumers import the module rather than reprocessing its source text.

```
import math.core;

int main() {
    return add(2, 3);
}
```

The compiler now understands module boundaries, dependencies, and visibility in a way that is impossible with textual inclusion.

### 1.2.2 Explicit Interfaces and Strong Boundaries

Modules expose only what is explicitly exported.

```
export module net.http;
```

```
export class Request {
public:
    void send();
};

void helper(); // not exported
```

The function `helper()` is completely invisible to importing code. This enforces encapsulation by default and prevents accidental growth of public APIs.

### 1.2.3 Build Scalability and Architectural Impact

Because module interfaces are compiled once and reused, build systems can avoid repeated parsing. This results in faster incremental builds, more effective parallel compilation, and more predictable build behavior.

More importantly, modules allow architecture to drive code structure. Dependencies become explicit. Boundaries are enforced by the language. Refactoring becomes safer because consumers cannot rely on hidden implementation details.

C++ Modules exist not as a superficial replacement for headers, but as a solution to a fundamental limitation in how large C++ systems have been built for decades. They provide the semantic foundation required for modern-scale, long-lived, and maintainable C++ software systems.

# Chapter 2

# What a C++ Module Really Is

## 2.1 Module Fundamentals

At its core, a C++ module is a language-level unit of program composition. Unlike headers, which rely on textual substitution, a module represents a semantically meaningful boundary that is understood and enforced by the compiler.
A module is identified by a unique name and consists of one or more translation units that collectively define its interface and implementation. The public surface of a module is compiled into a binary module interface, often referred to as a BMI, which can then be imported by other translation units.

```
export module math.core;

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

In this example, `math.core` is the module name. The `export` keyword indicates that the module is an interface unit and that the function `add` is part of the module's public API.
Only declarations explicitly marked for export become visible to code that imports the module. Everything else remains private by default, even though it resides in the same source file.

### 2.1.1 Binary Module Interfaces

When a module interface unit is compiled, the compiler produces a binary representation of the exported declarations. This binary module interface contains the semantic information needed by importing translation units, such as type definitions, function signatures, and template declarations.
Importing code does not reparse the original source text. Instead, it consumes the validated binary interface, which enables faster compilation and stronger guarantees about correctness.

### 2.1.2 Module Names and Structure

Module names are logical identifiers and do not correspond directly to file names or directory layouts. A single module may be implemented across multiple source files, and different build systems may organize module sources differently.
What matters is the declared module name, which serves as the importable unit of abstraction.

## 2.2 Visibility and Export Semantics

Visibility in modules is explicit and intentional. By default, declarations inside a module are not visible outside of it. Only declarations marked with `export` become part of the public interface.

```
export module math.internal;

export int multiply(int a, int b);

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

In this example, `multiply` is visible to importers, while `helper` remains completely hidden. This contrasts sharply with headers, where all declarations are exposed once included.

## 2.3 Modules vs Headers

The most important distinction between modules and headers lies in how they define and enforce boundaries.
Headers expose everything they contain. Once included, all declarations become part of the including translation unit, regardless of whether they are intended to be public or internal. This makes it difficult to maintain clean APIs and often leads to unintended dependencies.
Modules, by contrast, provide explicit control over visibility. Only what is exported is visible. Internal implementation details are protected by the language itself, not by convention or documentation.

### 2.3.1 Comparison by Example

Consider a traditional header-based interface:

```
// math.h
#pragma once

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

// internal helper
int normalize(int x);
```

All three functions are exposed to any file that includes this header, even if `normalize` is intended for internal use only.
Now compare this with a module-based interface:

```
export module math;

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

int normalize(int x);
```

Here, `normalize` is completely inaccessible to importing code. The compiler enforces this boundary, preventing accidental usage and protecting the integrity of the module's design.

### 2.3.2 Implications for Design

This explicit separation between interface and implementation has profound implications for software design. APIs become deliberate and stable. Refactoring internal code no longer risks breaking consumers. Large systems become easier to reason about because dependencies are visible and intentional.

A C++ module is therefore not merely a replacement for a header. It is a first-class architectural unit that enables stronger encapsulation, clearer interfaces, and more maintainable codebases at scale.

# Chapter 3

# Module Interface Units

## 3.1 Defining Interfaces

A module interface unit is the primary mechanism by which a C++ module defines its public contract. It represents the authoritative description of what a module offers to the rest of the program and serves as the single point of interaction between the module and its consumers. A module interface unit is declared using the `export module` syntax. Any declaration that is marked with `export` becomes part of the module's public API and is visible to translation units that import the module.

```
export module net.http;

export class Request {
public:
    void send();
};
```

In this example, the module named `net.http` exposes a single class, `Request`, along with its public member function `send`. The class definition, its layout, and its exported members

are part of the module interface and are available to all importing code.

### 3.1.1 What Belongs in an Interface Unit

The interface unit should contain only declarations that are intended to be used by consumers of the module. This typically includes:

- Public classes and structs
- Public function declarations
- Exported type aliases and enumerations
- Template declarations intended for external use

Implementation details, helper functions, and internal data structures should not appear in the interface unless they are explicitly part of the public contract.

### 3.1.2 Multiple Interface Units

A module may consist of multiple interface units. This allows large modules to expose logically separated interfaces while still sharing a common module name.

```
export module net.http;
export module net.http.client;
export module net.http.server;
```

This approach enables finer-grained organization of public APIs without sacrificing the benefits of modular compilation.

## 3.2 Encapsulation Benefits

One of the most significant advantages of module interface units is their impact on encapsulation. Unlike headers, which expose everything they contain, module interfaces expose only what is explicitly exported.

### 3.2.1 Hiding Implementation Details

Any declaration that is not marked with `export` remains invisible to importing code.

```
export module net.http;

export class Request {
public:
    void send();
};

void open_connection(); // not exported
```

In this example, `open_connection` is completely inaccessible to consumers of the module. This guarantees that internal mechanisms cannot be relied upon accidentally or intentionally by external code.

### 3.2.2 Stable APIs and Safer Refactoring

Because the public surface of a module is explicitly defined, it becomes much easier to maintain stable APIs. Internal changes can be made freely as long as the exported interface remains consistent.
Refactoring implementation details no longer risks breaking downstream code that might have been implicitly depending on internal declarations, as often happens with header-based designs.

### 3.2.3 Reduced Coupling

Consumers of a module depend only on the declarations they explicitly import. They cannot see or rely on unrelated internal components. This reduces coupling between modules and leads to cleaner, more maintainable architectures.
Module interface units therefore serve as both a technical and architectural boundary.
They define what a module is, what it guarantees, and how it can be used, while shielding consumers from unnecessary complexity and internal change.

# Chapter 4

# Module Implementation Units

## 4.1 Separating Implementation

Module implementation units are responsible for providing the concrete definitions of entities declared in a module interface unit. Unlike interface units, implementation units do not export symbols and do not define the public contract of the module. Their role is purely to supply behavior and internal logic.
An implementation unit is introduced using the `module` keyword followed by the name of the module it belongs to, without the `export` specifier.

```
module net.http;

void Request::send() {
    // implementation logic
}
```

This syntax explicitly associates the implementation with the `net.http` module and ensures that the definitions provided here apply to declarations previously exported by the module's interface unit.

### 4.1.1 Relationship to the Interface Unit

Every exported declaration that requires a definition must be defined in exactly one implementation unit or directly within the interface unit itself. Implementation units complete the module by supplying these definitions without expanding the module's public surface. This separation allows developers to reason about the public API and the internal behavior independently. Consumers interact only with the interface unit, while maintainers work primarily within implementation units.

### 4.1.2 Private Implementation Details

Implementation units are free to define internal helpers, data structures, and functions that are completely invisible outside the module.

```
module net.http;

namespace {
    void open_socket();
}

void Request::send() {
    open_socket();
}
```

These internal details are not accessible to importing code and cannot be accidentally relied upon. The compiler enforces this encapsulation, eliminating a common source of tight coupling in header-based designs.

### 4.1.3 Multiple Implementation Units

A module may be implemented across multiple implementation units. This allows large or complex modules to be split into manageable source files without exposing additional

interfaces.

```
// request.cpp
module net.http;

void Request::send() {
    // ...
}

// response.cpp
module net.http;

void Response::receive() {
    // ...
}
```

All of these units contribute to the same module and share access to its internal declarations, while still presenting a single, coherent public interface.

### 4.1.4 Maintainability and Compilation Benefits

Separating interface and implementation improves maintainability by localizing change. Modifications to implementation units do not affect consumers of the module as long as the interface remains stable.

From a build perspective, this separation reduces unnecessary recompilation. Changes to implementation files typically require recompiling only the module itself, not every translation unit that imports it.

Module implementation units therefore play a critical role in enabling scalable builds, clean architecture, and long-term maintainability in modern C++ systems.

# Chapter 5

# Import Semantics

## 5.1 Importing Modules

Modules are consumed using the `import` keyword. An import statement expresses a semantic dependency on a named module, rather than a textual dependency on a file.

```
import net.http;
```

When a translation unit imports a module, it gains access only to the declarations that were explicitly exported by that module's interface unit. No source text is copied, no macros are injected, and no hidden symbols become visible.

Importing a module instructs the compiler to load the module's binary interface and validate its contents against the current compilation context. If the module interface is incompatible or missing, the compilation fails immediately, rather than producing subtle errors later in the build or at link time.

### 5.1.1 Deterministic and Validated Dependencies

Because imports operate on compiled module interfaces, they are fully validated by the compiler. This eliminates many classes of errors that are common with textual inclusion, such as missing includes, order-dependent compilation, or accidental reliance on transitive headers. Each import statement declares a clear and intentional dependency. Build systems can analyze these dependencies accurately and schedule compilation steps accordingly.

### 5.1.2 Import Scope and Visibility

Imported modules affect only the translation unit in which they appear. They do not implicitly affect other files, nor do they leak symbols across unrelated compilation boundaries.

```
import math.core;

int compute() {
    return add(4, 5);
}
```

Here, only the exported interface of `math.core` is visible. No internal declarations or implementation details are accessible.

## 5.2 Comparison with `#include`

The difference between `import` and `#include` is fundamental.
The `#include` directive performs a textual copy of a header's contents into the current source file. The compiler then processes the resulting expanded text as if it were written directly in the file.
By contrast, importing a module does not reparse its contents. The compiler consumes a precompiled, semantically validated interface. This distinction has profound implications for correctness, performance, and maintainability.

### 5.2.1 Parsing and Compilation Costs

With headers, every translation unit reprocesses the same declarations, leading to redundant parsing and long build times. With modules, the interface is compiled once and reused.
This reduces:

- Compilation time
- Memory usage during compilation
- Sensitivity to include order
- Accidental dependency on transitive includes

### 5.2.2 Macro Isolation

Macros are one of the most problematic aspects of header-based designs. They ignore scope, leak across files, and can silently alter behavior.
Modules isolate macros by default. Macros defined inside a module interface do not automatically propagate to importing code, restoring predictability and locality to compilation.

# Chapter 6

# Header Units

## 6.1 Interoperating with Legacy Code

C++ Modules were designed with backward compatibility in mind. Recognizing that existing codebases rely heavily on headers, the standard introduces the concept of header units.
A header unit allows an existing header file to be treated as a module and imported using the `import` keyword.

```
import <vector>;
import <string>;
```

In this example, the standard library headers `<vector>` and `<string>` are imported as header units rather than included textually.

### 6.1.1 Purpose of Header Units

Header units serve as a bridge between traditional header-based code and modern module-based designs. They allow developers to benefit from some aspects of module semantics, such as reduced parsing and clearer dependency declarations, without rewriting existing libraries.

Header units make it possible to:

- Gradually adopt modules in existing projects
- Reduce reliance on textual inclusion
- Experiment with module-based builds incrementally

### 6.1.2 Limitations of Header Units

While header units provide useful interoperability, they do not offer all the benefits of true modules. Because they are derived from headers, they may still expose macros, inline definitions, and other artifacts that were never designed for strong encapsulation.
Header units should therefore be viewed as a transitional mechanism, not a final architectural solution.

### 6.1.3 Migration Strategy

A practical migration path often begins by importing standard library headers as header units, followed by converting internal libraries into true modules. Over time, header usage can be reduced and eventually eliminated for newly designed components.
Used correctly, header units enable a smooth and low-risk transition to C++ Modules, allowing teams to modernize their build and architecture without disrupting stable, existing code.

# Chapter 7

# Modules and the One Definition Rule

## 7.1 ODR Enforcement

The One Definition Rule (ODR) is one of the most fundamental and notoriously difficult aspects of the C++ language. It requires that every entity with external linkage has exactly one definition across the entire program, while allowing multiple compatible declarations.
In traditional header-based designs, ODR compliance is largely enforced by convention. Headers are guarded, inline specifiers are used carefully, and global objects are avoided or manually managed. Despite these practices, accidental ODR violations remain common, especially in large and long-lived codebases.
Modules fundamentally change this situation by enforcing the One Definition Rule at the module boundary.

### 7.1.1 ODR Problems in Header-Based Code

Consider a simple header:

```
// settings.h
```

```
#pragma once

int max_connections = 100;
```

When this header is included in multiple translation units, each unit receives its own definition of `max_connections`. This results in a violation of the One Definition Rule, often detected only at link time or, in some cases, not detected at all.
Even when developers attempt to mitigate this using `inline`, `static`, or `extern`, the responsibility remains manual and error-prone.

### 7.1.2 How Modules Improve ODR Safety

With modules, exported entities are compiled once as part of the module interface and implementation. The compiler has full visibility into where and how each exported definition is introduced.

```
export module config;

export int max_connections;
```

```
module config;

int max_connections = 100;
```

In this structure, the definition of `max_connections` exists in exactly one place. Importing translation units cannot accidentally introduce additional definitions, because they consume a validated binary interface rather than source text.

### 7.1.3 Early Error Detection

Because module interfaces are compiled independently, many ODR violations are detected earlier in the build process. Conflicts that would previously surface at link time are often caught during module compilation, reducing debugging time and improving build reliability.

Modules shift ODR enforcement from a fragile, convention-based model to a compiler-enforced guarantee.

# Chapter 8

# Build Systems and Toolchains

## 8.1 Compiler Support

C++ Modules are supported by all major modern C++ compilers, although the level of maturity and completeness varies.

### 8.1.1 GCC

GCC provides module support starting with C++20, with ongoing improvements in recent releases. GCC uses its own binary module interface format and requires explicit build rules to manage module dependencies.

### 8.1.2 Clang

Clang supports C++ Modules through its modular compilation pipeline and integrates closely with LLVM-based toolchains. Clang places strong emphasis on accurate dependency scanning and parallel compilation of module interfaces.

### 8.1.3 MSVC

MSVC offers one of the most mature implementations of C++ Modules, including deep integration with its build system and IDE tooling. It supports both standard library modules and user-defined modules with robust diagnostics and stable behavior.

## 8.2 Build System Responsibilities

Unlike header-based builds, module-based builds require build systems to understand compilation order explicitly. Module interface units must be compiled before any translation unit that imports them.
This means build systems must:

- Discover module dependencies accurately
- Schedule module interface compilation first
- Cache and reuse compiled module interfaces
- Rebuild only what is affected by interface changes

While this introduces additional complexity at the build system level, it enables far more predictable and scalable builds. Once properly configured, module-aware build systems provide faster incremental builds and clearer dependency management than traditional header-based approaches.
Modules therefore represent not only a language feature, but a shift in how C++ projects are built, organized, and maintained across different toolchains.

# Chapter 9

# Migrating Existing Codebases

## 9.1 Incremental Adoption

One of the most important design goals of C++ Modules was to allow incremental adoption. The language does not require existing codebases to be rewritten or converted wholesale. Instead, modules can be introduced gradually, alongside traditional headers, with minimal disruption.

Attempting a full migration in a single step is rarely practical. Large C++ systems often represent years of accumulated design decisions, third-party dependencies, and build system constraints. A successful migration strategy respects this reality.

### 9.1.1 Start with Leaf Components

The safest and most effective place to introduce modules is at the edges of the dependency graph. Leaf components are libraries or modules that:

- Have few or no dependencies

- Are widely used by other parts of the system
- Expose stable and well-defined APIs

Converting such components into modules yields immediate benefits: build times improve, interfaces become explicit, and downstream code can begin importing modules without affecting unrelated areas.

### 9.1.2 Isolate New Development

A common and effective strategy is to require that all new libraries and subsystems be written using modules from the outset. This prevents further growth of header-based complexity while allowing legacy code to remain stable.
Over time, the proportion of module-based code naturally increases, and the cost of migration is distributed across normal development cycles.

### 9.1.3 Avoid Early Large-Scale Refactoring

Early in the migration process, the goal should be containment, not perfection. Large-scale refactoring of core abstractions introduces unnecessary risk and often provides limited short-term benefit.
Instead:

- Preserve existing interfaces initially
- Replace headers with module interfaces where boundaries are clear
- Delay deep architectural changes until tooling and workflows are well understood

Modules are most effective when they reinforce good architecture. They should not be used as a forcing function for premature redesign.

### 9.1.4 Managing Mixed Codebases

During migration, it is common to operate with a mixed codebase that uses both headers and modules. This is not a failure state; it is an expected and supported phase.
Header units can be used to bridge the gap, allowing legacy headers to be imported while new code adopts module semantics. With careful dependency management, mixed codebases can remain stable, build efficiently, and evolve gradually.

# Chapter 10

# Design Guidelines for Modules

## 10.1 Module Granularity

Choosing the right granularity is one of the most important design decisions when working with modules. While modules allow strong encapsulation, they do not eliminate the need for thoughtful architectural boundaries.
Smaller, focused modules are generally easier to understand, maintain, and evolve. A module should represent a single, coherent responsibility rather than a broad collection of loosely related functionality.

### 10.1.1 Avoid Monolithic Modules

Large, monolithic modules tend to recreate the problems of large header files. They:

- Increase compile-time dependencies
- Make interfaces harder to reason about
- Encourage unrelated components to evolve together

Instead, prefer decomposing functionality into multiple modules with clear and narrow responsibilities. Composition through imports scales better than aggregation through export.

### 10.1.2 Align Modules with Architectural Boundaries

Modules should reflect architectural intent. If two components can evolve independently, they likely belong in separate modules. If a component has internal subsystems, consider whether they warrant separate modules or remain internal implementation details.
Well-aligned modules make dependency relationships visible and enforce them at the language level.

## 10.2 Export Discipline

Export discipline is essential to maintaining the long-term health of a module-based system. Every exported declaration becomes part of the module's public contract and carries an implicit commitment to stability.

### 10.2.1 Minimize the Public Surface Area

Export only what consumers truly need. Resist the temptation to export convenience functions, internal helper types, or implementation-specific abstractions.
A smaller public interface:

- Reduces coupling between modules
- Makes refactoring safer
- Simplifies documentation and reasoning
- Improves long-term maintainability

### 10.2.2 Treat Exports as Contracts

Once a declaration is exported, changing or removing it may affect multiple consumers.
Exports should therefore be treated as explicit contracts, not implementation details.
Careful export discipline encourages deliberate API design and prevents the gradual erosion of architectural boundaries—a common failure mode in large header-based systems.
When used thoughtfully, modules not only improve build performance but also reinforce architectural clarity and long-term system integrity.

# Chapter 11

# Performance Implications

## 11.1 Build-Time Performance

One of the primary motivations for introducing C++ Modules is the significant improvement in build-time performance. Traditional header-based compilation requires the compiler to repeatedly parse the same declarations across many translation units. In large projects, this redundant work dominates build times and slows developer iteration.

Modules address this issue by compiling interfaces once and reusing the result across the entire build. When a module interface unit is compiled, the compiler produces a binary module interface that captures all necessary semantic information. Subsequent imports of the module reuse this compiled representation rather than reprocessing source text.

This approach yields several concrete benefits:

- Dramatically reduced parsing time
- Faster incremental builds when implementations change
- Improved parallelism in compilation

- More predictable and stable build performance

For example, consider a project with hundreds of translation units that all depend on a common utility library. With headers, any change to a frequently included header can trigger a full rebuild. With modules, changes confined to implementation units often require recompiling only the module itself.

### 11.1.1 Impact on Incremental Builds

Incremental builds benefit especially from modules. When an implementation unit changes but the module interface remains unchanged, importing translation units do not need to be recompiled. This allows developers to iterate on internal logic without paying the cost of recompiling large portions of the system.
Over time, this leads to faster feedback loops, higher productivity, and greater willingness to refactor internal code.

## 11.2 Runtime Performance

C++ Modules do not directly affect runtime performance. They introduce no additional abstraction layers, runtime indirection, or execution overhead. The generated machine code for a program using modules is equivalent to that of a well-structured header-based program. This is an intentional design decision. Modules adhere to C++'s zero-overhead principle: they improve structure, correctness, and build efficiency without changing runtime behavior.
Any performance improvements observed at runtime are therefore indirect. They result from better architecture, clearer interfaces, and increased developer confidence to optimize and refactor code safely.

# Chapter 12

# Common Pitfalls

While modules provide powerful benefits, they also introduce new design considerations. Misunderstanding their role or applying them incorrectly can limit their effectiveness.

## Treating Modules as Headers

One of the most common mistakes is treating modules as a direct replacement for headers without changing design habits. Simply converting headers into module interface units without improving encapsulation or dependency structure yields limited benefit.
Modules are most effective when interfaces are intentionally designed, not mechanically translated.

## Exporting Implementation Details

Exporting too many declarations undermines the encapsulation benefits of modules. Every exported symbol becomes part of the public contract and increases coupling between components.

Internal helper functions, data structures, and experimental APIs should remain unexported whenever possible.

## Creating Cyclic Module Dependencies

Although modules make dependencies explicit, they do not eliminate the possibility of poor architectural decisions. Cyclic dependencies between modules complicate build ordering and often signal deeper design problems.
A well-designed module graph should be largely acyclic and reflect clear layering within the system.

## Overusing Header Units

Header units are a valuable migration tool, but they are not a substitute for true modules. Overreliance on header units can perpetuate many of the same issues associated with traditional headers, including macro leakage and weak encapsulation.
Header units should be viewed as a transitional mechanism, not a final state.

## Summary

C++ Modules offer substantial improvements in build scalability and architectural clarity without affecting runtime performance. When used thoughtfully and with discipline, they enable faster builds, safer refactoring, and more maintainable systems. Avoiding common pitfalls ensures that these benefits are fully realized over the long term.

# Conclusion

C++ Modules represent one of the most significant structural improvements to the C++ language since the introduction of templates and the Standard Library. They address a long-standing limitation in how C++ programs are composed, compiled, and maintained at scale. For decades, developers have relied on conventions, discipline, and tooling workarounds to manage the weaknesses of the header-based compilation model. While these techniques can mitigate some problems, they cannot eliminate the fundamental issues inherent in textual inclusion. As systems grow larger and more complex, these issues increasingly constrain architecture, slow development, and raise maintenance costs.

Modules provide a language-level solution. By replacing textual inclusion with semantic import, they introduce explicit interfaces, enforce clear boundaries, and allow the compiler to reason about dependencies with precision. This enables large systems to scale without sacrificing performance, determinism, or control—the core values that have always defined C++.

Adopting modules is not a purely mechanical exercise. It requires careful planning, deliberate interface design, and a clear understanding of existing architectural boundaries. Early adoption should focus on incremental integration, stable APIs, and realistic migration strategies rather than wholesale rewrites.

The long-term benefits, however, are substantial. Teams that adopt modules gain cleaner and more intentional architectures, dramatically faster and more predictable builds, and codebases that are easier to reason about, refactor, and extend over time.

C++ Modules are not a trend or an optional enhancement. They are the foundation upon which modern, long-lived C++ systems can be built and sustained with confidence for the decades ahead.

# Appendices

## Glossary

This glossary summarizes the most important terms and concepts related to C++ Modules as used throughout this booklet. The definitions are intentionally concise but precise, reflecting their practical meaning in real-world C++ systems.

| | |
|---|---|
| Module Interface Unit | A translation unit that defines the public interface of a module. It is introduced using the `export module` syntax and contains all declarations that are explicitly exported for use by importing code. The interface unit is compiled into a binary module interface and serves as the authoritative contract of the module. |
| Module Implementation Unit | A translation unit that provides definitions for declarations declared in a module interface unit. Implementation units are associated with a module using the `module` keyword and do not export symbols. They contain internal logic, helper functions, and implementation details that are hidden from consumers. |

| | |
|---|---|
| Header Unit | A traditional C or C++ header that is treated as a module and imported using the `import` keyword. Header units enable interoperability with existing header-based code and support incremental migration to modules, but they do not provide the full encapsulation benefits of true modules. |
| Binary Module Interface (BMI) | The compiled, compiler-specific binary representation of a module's interface. A BMI contains semantic information such as type definitions, function signatures, and template declarations. It is consumed by importing translation units instead of reparsing source text, enabling faster and more reliable compilation. |
| Module Name | A logical identifier that uniquely names a module and is used in `import` statements. Module names are independent of file names and directory structure and represent the unit of composition and dependency in a module-based system. |
| Exported Declaration | Any declaration marked with the `export` keyword in a module interface unit. Exported declarations form the public API of a module and are visible to importing code. They should be treated as stable contracts. |
| Import Statement | A statement using the `import` keyword that declares a semantic dependency on a module or header unit. Import statements replace textual inclusion and are validated by the compiler. |

| | |
|---|---|
| One Definition Rule (ODR) | A core C++ rule requiring that entities with external linkage have exactly one definition across the entire program. Modules enforce ODR at module boundaries, reducing accidental violations common in header-based designs. |

# References

The material presented in this booklet is based on a combination of formal language specifications, long-established best practices, and real-world experience gained from building and maintaining large-scale C++ systems. The following references represent the primary sources of concepts, terminology, and guidance reflected throughout the text.

- **ISO/IEC C++20 and C++23 Standards**
  The official language standards defining the syntax, semantics, and behavior of C++ Modules, including module interface units, implementation units, import semantics, header units, and rules related to visibility and the One Definition Rule.

- **C++ Core Guidelines**
  A curated set of modern C++ design and coding guidelines that emphasize clarity, safety, performance, and maintainability. The principles behind explicit interfaces, encapsulation, and dependency management align closely with the goals of C++ Modules.

- **Compiler Documentation (GCC, Clang, MSVC)**
  Vendor-specific documentation describing the practical implementation details of C++ Modules, including compiler flags, module formats, build integration, and known limitations. These resources are essential for understanding real-world module support across different toolchains.

- **Industrial C++ Architecture Case Studies**
  Published experiences and internal engineering reports from large-scale C++ projects. These case studies illustrate the architectural challenges of header-based systems and the practical motivations behind adopting modules in production environments.

- **Build System and Toolchain Manuals**
  Documentation for modern build systems and toolchains that support module-aware builds. These manuals explain dependency discovery, compilation ordering, and incremental build strategies required to integrate C++ Modules effectively.

Together, these references provide the technical foundation for the concepts discussed in this booklet and reflect current, industry-tested best practices for using C++ Modules in modern software systems.