The Lateralus Module System

bad-antics | October 2024 | Language Design

Abstract

This paper describes the design and implementation of the Lateralus module system. Topics include module structure, visibility and encapsulation, imports and re-exports, dependency management, separate compilation, module initialization, conditional compilation, and name resolution.

1 Introduction

The Lateralus module system organizes code into hierarchical namespaces, controls visibility, and manages dependencies between compilation units. This paper describes the design and implementation of the module system.

Design goals: separate compilation (modules compile independently), encapsulation (implementation details are hidden), namespace management (avoid name collisions), and dependency tracking (explicit imports and exports).

The module system supports: file-based modules (one file per module), inline modules (modules within files), re-exports (exposing submodule items), and conditional compilation (platform-specific modules).

2 Module Structure

A crate is the top-level compilation unit. The crate root is main.lat (for executables) or lib.lat (for libraries). All other modules are descendants of the crate root.

File modules: each .lat file defines a module. The file name determines the module name. The file foo.lat in the crate root defines module foo.

Directory modules: a directory foo/ with a mod.lat file defines module foo. Files within the directory define submodules: foo/bar.lat defines foo::bar.

Inline modules: mod name { items } defines a module within the current file. Inline modules share the file with their parent. Inline modules are useful for small, related groupings.

Module paths: items are referenced by path. crate::module::item starts from the crate root. self::item refers to the current module. super::item refers to the parent.

3 Visibility

Items are private by default. Private items are accessible only within the defining module and its descendants. Private items are hidden from sibling and parent modules.

pub makes an item visible to any module. pub(crate) restricts visibility to the current crate. pub(super)

restricts to the parent module. pub(in path) restricts to the specified module.

Struct field visibility: struct fields are private by default. pub fields are accessible outside the module. A struct with private fields cannot be constructed outside the module.

Enum variant visibility: enum variants inherit the enum's visibility. If an enum is pub, all variants are pub. Individual variant visibility cannot be overridden.

Impl block visibility: methods follow the same visibility rules as other items. A pub method on a pub struct is accessible everywhere. A private method is accessible only within the module.

4 Imports and Re-exports

Use declarations bring items into scope. use module::item; makes item available without the module prefix. use module::item as alias; creates an alias.

Glob imports: use module::*; imports all public items from the module. Glob imports are convenient for preludes but should be used sparingly to avoid name collisions.

Nested imports: use module::{item1, item2, item3}; imports multiple items. use module::{self, item}; imports both the module and an item.

Re-exports: pub use module::item; re-exports an item from a submodule. Re-exports create a public API surface without exposing the internal module structure.

Path re-exports: pub use module::*; re-exports all public items. This pattern is used to create facade modules that aggregate items from multiple submodules.

5 Dependency Management

External dependencies are declared in the crate manifest (lateralus.toml). Each dependency specifies: name, version constraint, optional features, and source (registry, git, or local path).

Version resolution: the package manager selects versions satisfying all constraints. Semver compatibility: ^1.2.3 allows 1.2.3 to 1.x.x. Exact: =1.2.3. Range: >=1.0, <2.0.

Lock file: lateralus.lock records the exact versions selected. The lock file ensures reproducible builds. Updating dependencies requires explicit lateralus-pkg update.

Feature flags: dependencies can expose optional features. Features enable conditional compilation and optional functionality. Enabling a feature may enable transitive dependencies.

6 Separate Compilation

Each module is compiled independently. The compiler produces a module interface file (.lmi) containing: public type signatures, function signatures, and constant values.

Incremental compilation: the build system tracks dependencies between modules. When a module changes, only the module and its dependents are recompiled.

Compilation order: modules are compiled in topological order of dependencies. Circular dependencies between modules are not allowed and produce compile errors.

Interface stability: a module's interface file is compared with the previous build. If the interface is unchanged, dependents are not recompiled even if the implementation changed.

7 Module Initialization

Static initialization: constants and statics in each module are initialized before main() runs. Initialization order follows the module dependency graph.

Lazy initialization: lazy_static! values are initialized on first access. Lazy initialization avoids ordering issues and reduces startup time for rarely-used modules.

Initialization errors: if a module's initialization panics, the program aborts. Initialization code should be minimal and infallible.

8 Conditional Compilation

cfg attributes enable platform-specific code: #[cfg(target_os = 'linux')]. cfg conditions include: target_os, target_arch, feature, debug_assertions, and custom conditions.

cfg_attr applies attributes conditionally: #[cfg_attr(feature = 'serde', derive(Serialize, Deserialize))]. The attribute is applied only when the condition is true.

Platform-specific modules: mod platform; declares a module. The compiler selects platform/linux.lat or platform/macos.lat based on the target.

9 Name Resolution

Name resolution resolves identifiers to their definitions. The resolver processes: local variables, function parameters, module items, imported items, and prelude items.

Shadowing: a local variable can shadow a module item. Inner scopes shadow outer scopes. Shadowing is allowed but generates a warning when the types differ.

Ambiguity: if an identifier resolves to multiple items (from different glob imports), a compile error is reported. The error suggests qualifying the name with the module path.

10 Conclusion

The Lateralus module system provides hierarchical organization, controlled visibility, separate

compilation, and reproducible dependency management. The design balances encapsulation with usability.

2.1 Crate Types

Binary crates: compiled to executables. The crate root must contain a main() function. Binary crates cannot be used as dependencies.

Library crates: compiled to reusable libraries. The crate root is lib.lat. Library crates export a public API for other crates to consume.

Proc-macro crates: contain procedural macros. Proc-macro crates are compiled for the host architecture and run during compilation of dependent crates.

Build script crates: build.lat runs before the main compilation. Build scripts generate code, detect system libraries, and configure compilation options.

Test crates: #[cfg(test)] mod tests { } defines test-only modules. Test modules have access to private items in their parent module.

Benchmark crates: bench/ contains benchmark programs. Benchmark crates measure performance and detect regressions. They share dependencies with the main crate.

Example crates: examples/ contains example programs. Example crates demonstrate usage patterns and serve as integration tests. Each example is a separate binary.

Workspace crates: a workspace contains multiple crates that share a build directory, lock file, and dependency resolution. Workspaces enable monorepo development.

Virtual workspace: a workspace without a root crate. Virtual workspaces are containers for multiple related crates without a single top-level entry point.

Crate naming: crate names must be unique within a registry. Names use lowercase letters, digits, and hyphens. Underscores in code reference hyphens in crate names.

3.1 Visibility Rules Detail

Privacy boundary: the module is the privacy boundary. Items in a module can access all items in the same module and its descendants, regardless of visibility.

Ancestor access: a module can access its ancestors' public and pub(crate) items. Private items in ancestors are not accessible.

Sibling access: sibling modules can access each other's public and pub(crate) items. Private items in siblings are not accessible.

Struct constructor visibility: if any field is private, the struct cannot be constructed outside the module. Use a pub fn new() constructor to provide controlled construction.

Visibility and traits: a trait method's visibility is determined by the trait's visibility. Implementing a public trait requires public methods. Implementing a private trait can use private methods.

Visibility and generics: a public generic type can have private type parameters in its implementation. Users cannot name the private type but can use the public API.

Re-export visibility escalation: pub use private_module::Item; makes a private module's item public. This is the idiomatic way to create a public API from private implementation.

Visibility lint: the compiler warns about: public items that are not reachable (dead public items), and private items that could be public (suggestions).

Visibility documentation: pub items appear in generated documentation. Private items are hidden by default. #[doc(hidden)] hides public items from documentation.

Visibility and FFI: extern functions are always public in the linker sense. Lateralus visibility only affects Lateralus-level access. The linker sees all non-private symbols.

4.1 Import Resolution Algorithm

Import resolution runs after parsing and before type checking. The resolver processes imports in topological order of module dependencies.

Single import resolution: use path::item; looks up the path, then looks up item in the resolved module. The item must be visible from the importing module.

Glob import resolution: use path::*; imports all public items from the resolved module. Glob imports have lower priority than single imports (shadowed by explicit imports).

Ambiguity detection: if two glob imports provide items with the same name, the name is ambiguous. Using the ambiguous name is a compile error.

Import cycle detection: imports cannot form cycles. use a::b; in module a where b uses a::c; is allowed (not a cycle). use a; in module b and use b; in module a is a cycle.

Lazy resolution: imports can refer to items defined later in the module. The resolver processes all imports before checking item definitions.

Type import vs value import: imports distinguish between type namespace and value namespace. A module can have both struct Foo (type) and fn Foo (value) in scope.

Use tree flattening: use module::{a, b::{c, d}}; is flattened to: use module::a; use module::b::c; use module::b::d; Flattening simplifies resolution.

Import diagnostics: unresolved imports produce error messages with: the import path, the unresolved segment, and suggestions (did you mean?).

Import ordering: by convention, imports are grouped as: standard library, external crates, local modules. Import groups are separated by blank lines.

5.1 Dependency Resolution Algorithm

The resolver builds a dependency graph. Each node is a (crate, version) pair. Edges represent dependency relationships. The graph must be a DAG (no circular dependencies).

Version selection: for each dependency, the resolver selects the newest version satisfying all constraints. If constraints conflict, the resolver reports an error.

Minimal version selection (optional): selects the oldest compatible version instead of the newest. Minimal version selection ensures that Cargo.toml specifies the true minimum requirements.

Feature unification: if multiple crates depend on the same crate with different features, features are unified (all requested features are enabled).

Optional dependencies: dependencies with optional = true are only included when a feature enables them. Optional dependencies reduce binary size and compilation time.

Platform-specific dependencies: [target.'cfg(target_os = 'linux')'.dependencies] includes dependencies only on matching platforms.

Dev dependencies: [dev-dependencies] are only used for tests and benchmarks. Dev dependencies are not included in library builds.

Build dependencies: [build-dependencies] are used by build scripts. Build dependencies are compiled for the host architecture, not the target.

Dependency audit: lateralus-audit checks dependencies for known vulnerabilities. The audit database is updated regularly. CI pipelines should include audit checks.

License compatibility: the resolver warns about license incompatibilities between dependencies. Strict mode rejects incompatible licenses.

6.1 Module Interface Files

Interface file format: the .lmi file is a binary format containing: type definitions, function signatures, constant values, and re-export mappings.

Type signatures: public struct and enum definitions are fully described. Private fields are omitted. Generic constraints are included.

Function signatures: public function signatures include: parameter names and types, return type, generic parameters, and trait bounds. Function bodies are not included.

Constant values: public constants are included by value. The value is embedded in the interface file. Dependent modules use the value directly.

Interface versioning: the interface file includes a hash of the public API. If the hash changes, dependent modules must be recompiled. Implementation changes do not change the hash.

Interface generation: the compiler generates the interface file during compilation. The interface is

written atomically (to a temporary file, then renamed) to avoid partial writes.

Interface caching: the build system caches interface files. If the source module has not changed, the cached interface is used. This avoids redundant compilation.

Cross-compilation interfaces: interface files are architecture-independent for most types. Architecture-dependent types (pointer size, alignment) are parametric in the interface.

Interface diffing: the build tool can diff two interface files to show API changes. This is useful for: version bumps, code review, and API stability checking.

Interface documentation: the interface file includes documentation comments. Tools can generate API documentation from the interface without accessing the source.

7.1 Initialization Order

Static initialization order follows the module dependency graph. If module A depends on module B, B's statics are initialized before A's.

Within a module, statics are initialized in declaration order. Constants are evaluated at compile time and do not participate in runtime initialization.

Initialization detection: the compiler detects statics that depend on other statics and orders initialization accordingly.

Circular static dependency: if static A's initializer references static B and B's references A, the compiler reports a circular dependency error.

Thread-local statics: thread_local! statics are initialized on first access by each thread. Thread-local initialization is independent of module initialization.

Initialization timing: all module initialization completes before main() is called. The initialization order is deterministic across builds.

Initialization failure: if a static initializer panics, the program aborts immediately. No cleanup is performed. Initializers should be simple and infallible.

Lazy initialization: lazy_static! delays initialization to first access. The initialization function runs exactly once, even with concurrent access.

Global allocator initialization: the global allocator is initialized before any other static. This ensures that heap allocation is available during other initializations.

Initialization profiling: the build tool can measure initialization time for each module. Long initialization times indicate: expensive computations, I/O in initializers, or too many statics.

8.1 Conditional Compilation Details

cfg conditions: target_os (linux, macos, windows), target_arch (x86_64, aarch64, riscv64), target_env

(gnu, musl, msvc), target_endian (little, big), target_pointer_width (32, 64).

Feature conditions: #[cfg(feature = 'name')] tests whether a feature is enabled. Features are enabled by: the crate manifest, dependent crate features, or command-line flags.

Debug conditions: #[cfg(debug_assertions)] is true in debug builds. #[cfg(not(debug_assertions))] is true in release builds. Debug conditions control: assertions, bounds checks, and debug output.

Custom conditions: the build script can set custom cfg values: println!('cargo:rustc-cfg=has_avx2'). Custom conditions enable: hardware detection, OS version detection, and build configuration.

cfg combinators: #[cfg(all(target_os = 'linux', target_arch = 'x86_64'))]. #[cfg(any(target_os = 'linux', target_os = 'macos'))]. #[cfg(not(target_os = 'windows'))].

cfg in expressions: if cfg!(target_os = 'linux') { linux_code() } else { generic_code() }. The cfg! macro evaluates to true or false at compile time.

Platform-specific modules: the compiler automatically selects platform-specific source files based on cfg conditions. src/sys/linux.lat vs src/sys/macos.lat.

cfg and testing: #[cfg(test)] marks test-only code. Test modules can access private items. cfg(test) code is stripped from release builds.

cfg documentation: platform-specific items are documented with their cfg conditions. Documentation tools show: which platforms support each item.

cfg linting: the compiler warns about: impossible cfg conditions (no platform matches), always-true conditions (condition is satisfied on all platforms), and misspelled condition values.

9.1 Name Resolution Algorithm

The name resolver operates in three phases: macro expansion (expands macros that may introduce new items), import resolution (resolves use declarations), and item resolution (resolves identifiers in code).

Scope chain: name lookup searches: the current block, enclosing blocks, the current function, the current module, imported items, and the prelude. The first match wins.

Macro expansion interleaving: macro expansion may introduce new items and imports. The resolver re-runs import resolution after each round of macro expansion.

Forward references: within a module, items can reference items defined later. The resolver processes all items before checking references.

Namespace separation: types, values, and macros occupy separate namespaces. A module can have: struct Foo (type namespace), fn Foo (value namespace), and macro Foo (macro namespace).

Qualified paths: Type::associated_item resolves the type, then looks up the associated item. Trait::method resolves the trait, then looks up the method.

Method resolution: method calls search: inherent methods (on the type), trait methods (on implemented traits), and deref methods (on the deref target). The first match wins.

Trait method disambiguation: if multiple traits provide a method with the same name, the caller must disambiguate: <Type as Trait>::method().

Glob import shadowing: explicit imports shadow glob imports. If module A provides item X (via glob) and module B provides item X (via explicit import), B's X is used.

Resolution caching: the resolver caches resolved paths. Subsequent references to the same path use the cached result. Caching reduces resolution time for large modules.

2.2 Module File Layout

Flat layout: all modules in the same directory. src/main.lat, src/parser.lat, src/codegen.lat. Simple projects use flat layout.

Hierarchical layout: modules organized in directories. src/compiler/parser.lat, src/compiler/codegen.lat. Each directory has mod.lat for the directory module.

Mixed layout: top-level modules use files, complex modules use directories. This balances simplicity and organization.

Test layout: tests/ contains integration tests. Each file in tests/ is a separate test binary. Unit tests are in-module: #[cfg(test)] mod tests { }.

Benchmark layout: benches/ contains benchmark programs. Each file is a separate benchmark binary. Benchmarks use the criterion crate for statistical analysis.

Example layout: examples/ contains example programs. Each file is a separate binary. Examples demonstrate library usage and serve as documentation.

Build script layout: build.lat in the crate root is the build script. Complex build scripts use a build/ directory with multiple helper files.

Documentation layout: doc/ contains supplementary documentation. README.md in the crate root provides the crate overview. Each module can have a module-level doc comment.

Module naming conventions: module names use snake_case. Crate names use kebab-case (hyphens). The compiler converts hyphens to underscores for module names.

Path length limits: the compiler supports module paths up to 128 segments. File paths up to 4096 characters. These limits accommodate very large projects.

3.2 Visibility and Encapsulation Patterns

Newtype encapsulation: pub struct Id(i64); hides the inner type. Access to the inner value is through methods only. This prevents: accidental misuse, invalid values, and breaking changes.

Module-level encapsulation: group related types and functions in a module. Make only the API types and functions public. Implementation details are private.

Facade pattern: pub use submodule::Type; re-exports types from submodules. Users see a flat namespace. Implementation uses a deep module hierarchy.

Sealed trait pattern: define a trait and a private supertrait. External code cannot implement the sealed trait because the supertrait is private.

Builder pattern: pub struct Config { fields } with private fields. pub fn new() -> ConfigBuilder starts building. pub fn build(self) -> Config validates and constructs.

Type state pattern: struct Locked; struct Unlocked; prevent invalid state transitions at compile time. pub fn unlock(self) -> Door[Unlocked] moves to the unlocked state.

Extension trait pattern: define a trait with methods for existing types. pub trait VecExt { fn sorted(self) -> Self; } adds methods to Vec without modifying Vec.

Private module with public re-export: the implementation lives in a private module. The parent module re-exports the public API. This separates API from implementation.

Test-only visibility: #[cfg(test)] pub fn internal_state() -> State exposes private state for testing. Test-only visibility is stripped from release builds.

Deprecation path: pub fn old_api() is deprecated. pub fn new_api() replaces it. The old API delegates to the new API. Users are warned to migrate.

4.2 Prelude and Standard Imports

The standard prelude is imported into every module. The prelude contains frequently used items to avoid repetitive imports.

Prelude types: Option (Some, None), Result (Ok, Err), Box, Vec, String, str. These types are used in nearly every Lateralus program.

Prelude traits: Clone, Copy, Debug, Default, Display, Drop, Eq, PartialEq, Hash, Ord, PartialOrd, Iterator, IntoIterator, From, Into, AsRef, AsMut.

Prelude functions: println!, eprintln!, format!, panic!, assert!, assert_eq!, assert_ne!, dbg!, todo!, unimplemented!, unreachable!.

Prelude re-exports: the prelude re-exports items from: std::option, std::result, std::vec, std::string, std::boxed, and std::fmt.

Custom preludes: libraries can define a prelude module: pub mod prelude { pub use ...; }. Users import with: use library::prelude::*;

no_std prelude: programs without the standard library (#![no_std]) get a minimal prelude from the core library. The core prelude contains: Option, Result, and basic traits.

Edition-specific preludes: each language edition can add items to the prelude. New items are added in new editions to avoid breaking existing code.

Prelude conflicts: if a user-defined item conflicts with a prelude item, the user-defined item takes precedence. Prelude items are shadowed.

Prelude documentation: the prelude is documented in the standard library reference. Each item's documentation explains why it is included in the prelude.

5.2 Workspace Management

Workspace definition: a [workspace] section in the root lateralus.toml lists member crates. Members share: build output, lock file, and dependency resolution.

Shared dependencies: workspace members share dependency versions. A dependency used by multiple members is compiled once. Sharing reduces compilation time.

Workspace inheritance: member crates inherit workspace-level settings: author, license, edition, and repository. Members can override inherited settings.

Inter-workspace dependencies: workspace members can depend on each other using path dependencies. The build system resolves inter-workspace dependencies correctly.

Workspace commands: lateralus-build --workspace builds all members. lateralus-test --workspace tests all members. Commands can target specific members with --package.

Workspace CI: CI pipelines for workspaces test all members. Change detection identifies affected members and tests only those. Full builds run on release branches.

Monorepo strategy: large projects use workspaces as monorepos. Benefits: shared tooling, atomic cross-crate changes, unified versioning, and simplified CI.

Workspace publishing: members can be published independently to the registry. The workspace lockfile ensures consistent dependency versions across published members.

Virtual workspace: a workspace without a root package. The root lateralus.toml only contains [workspace]. Used when: all crates are equal members without a primary package.

Workspace policies: workspace-level configuration enforces: minimum supported Lateralus version, required features, banned dependencies, and code quality standards.

6.2 Incremental Compilation Details

Dependency graph: the build system maintains a directed graph of module dependencies. An edge from A to B means A imports B. Changes to B trigger recompilation of A.

File hash tracking: each source file's content hash is stored. If the hash has not changed since the last build, the module is skipped.

Interface hash tracking: the module interface hash determines whether dependents need recompilation. Implementation-only changes do not affect the interface hash.

Fine-grained invalidation: the build system tracks which items in a module are used by each dependent. If only unused items changed, the dependent is not recompiled.

Compilation caching: compiled object files are cached by (source hash, compiler version, compilation flags). Switching between branches reuses cached objects.

Parallel compilation: independent modules are compiled in parallel. The build system uses the dependency graph to determine which modules can compile concurrently.

Compilation progress: the build tool displays: total modules, compiled modules, cached modules, and estimated time remaining. Progress feedback improves developer experience.

Build profiling: the build tool records compilation time for each module. Long-compilation modules are identified for optimization. The profile is stored for trend analysis.

Clean builds: lateralus-clean removes all cached objects. Clean builds are needed after: compiler upgrades, toolchain changes, or suspected cache corruption.

Distributed compilation: the build tool supports distributing compilation across multiple machines. Each machine compiles a subset of modules. Results are collected and linked.

7.2 Global State Management

Global variables: static mut NAME: Type = value; declares a mutable global. Accessing mutable globals requires unsafe because of potential data races.

Thread-safe globals: static NAME: Mutex[Type] = Mutex::new(value); provides thread-safe global state. Accessing through lock() is safe.

Once-initialized globals: static NAME: OnceCell[Type] = OnceCell::new(); is initialized once via set() or get_or_init(). Subsequent reads return the initialized value.

Thread-local globals: thread_local! { static NAME: Cell[Type] = Cell::new(value); } provides per-thread state. Thread-local access is efficient (no synchronization).

Global allocator: #[global_allocator] static ALLOC: MyAllocator = MyAllocator; replaces the default memory allocator. Custom allocators implement GlobalAlloc.

Global panic handler: #[panic_handler] fn panic(info: &PanicInfo) -> ! { } defines the panic handler for no_std programs.

Global logger: static LOGGER: OnceCell[Logger] = OnceCell::new(); provides a global logging facility. The logger is initialized once during startup.

Configuration globals: static CONFIG: Lazy[Config] = Lazy::new(|| load_config()); loads configuration lazily. The configuration is immutable after initialization.

Global state testing: tests that modify global state must be serialized. The test framework provides: test isolation, state reset, and mutex-based serialization.

Global state guidelines: minimize global state. Prefer dependency injection. When global state is necessary, use thread-safe types. Document all global state.

8.2 Platform Abstraction Layer

The standard library uses conditional compilation to provide platform-specific implementations behind a common interface.

File system abstraction: std::fs provides File, OpenOptions, Metadata. The implementation uses: Linux system calls on Linux, Win32 API on Windows, POSIX on macOS.

Network abstraction: std::net provides TcpStream, TcpListener, UdpSocket. The implementation uses: epoll on Linux, kqueue on macOS, IOCP on Windows.

Thread abstraction: std::thread provides Thread, spawn, sleep, yield. The implementation uses: pthreads on Unix, CreateThread on Windows.

Synchronization abstraction: std::sync provides Mutex, RwLock, Condvar, Barrier. The implementation uses: futex on Linux, os_unfair_lock on macOS, SRWLOCK on Windows.

Process abstraction: std::process provides Command, spawn, wait, kill. The implementation uses: fork/exec on Unix, CreateProcess on Windows.

Time abstraction: std::time provides Instant (monotonic), SystemTime (wall clock), Duration. Implementations use: clock_gettime on Linux, mach_absolute_time on macOS.

Environment abstraction: std::env provides: args (command-line arguments), vars (environment variables), current_dir, home_dir, temp_dir.

Path abstraction: std::path provides Path, PathBuf. Path handling accounts for: platform-specific separators, drive letters (Windows), and UNC paths.

Platform detection: std::env::consts provides: OS (operating system name), ARCH (CPU architecture), FAMILY (unix or windows), EXE_SUFFIX, DLL_SUFFIX.

9.2 Macro System Integration

Declarative macros: macro_rules! name { (pattern) => { expansion } }. Macros operate on syntax tokens. Macros are expanded before name resolution.

Macro scope: macros defined with #[macro_export] are available crate-wide. Non-exported macros are local to their defining module.

Macro hygiene: macros introduce names that do not collide with names in the expansion context. Hygienic macros prevent accidental name capture.

Macro debugging: #[trace_macros] prints macro expansions during compilation. cargo expand shows the fully expanded source code.

Procedural macros: proc-macro crates define macros as functions. Input: TokenStream. Output: TokenStream. Proc macros enable: derive macros, attribute macros, and function-like macros.

Derive macros: #[derive(MyTrait)] expands to an impl block. The derive macro receives the struct/enum definition and generates the implementation.

Attribute macros: #[my_attr] transforms the annotated item. Attribute macros can: add fields, modify methods, and generate additional items.

Function-like macros: my_macro!(input) transforms arbitrary token input. Function-like macros are the most flexible but hardest to debug.

Macro and modules: macros respect module boundaries. A macro defined in module A is not available in module B unless exported and imported.

Macro best practices: prefer functions over macros. Use macros for: compile-time code generation, DSLs, and repetitive patterns. Document macro syntax clearly.

3.3 API Design Guidelines

Naming conventions: types use PascalCase. Functions and variables use snake_case. Constants use SCREAMING_SNAKE_CASE. Lifetimes use lowercase ('a). Type parameters use single uppercase (T).

Method naming: constructors use new(). Conversion methods use to_type(), into_type(), as_type(). Predicate methods use is_condition(). Accessor methods use get_field() or field().

Error handling conventions: return Result for recoverable errors. Panic for logic bugs. Use Option for absent values. Provide both panicking (unwrap) and safe (get) variants.

Iterator conventions: implement IntoIterator for collection types. Provide iter() for shared iteration and iter_mut() for mutable iteration.

Builder conventions: use the builder pattern for complex construction. Builders consume self and return Self for chaining. The build() method consumes the builder and returns the target type.

Trait design: keep traits focused (single responsibility). Provide default implementations where sensible. Use associated types to reduce generic parameters.

Deprecation: use #[deprecated(since = '1.2', note = 'use new_api()')] to deprecate items. Maintain deprecated items for at least one major version.

Documentation: every public item should have a doc comment. Doc comments include: purpose, parameters, return value, errors, panics, examples, and safety considerations.

Stability: mark experimental APIs with #[unstable]. Stabilize after sufficient usage and feedback.

Follow semantic versioning for API changes.

Performance documentation: document performance characteristics. Big-O complexity for algorithms. Allocation behavior for data structures. Cache behavior for critical paths.

5.3 Registry and Distribution

Package registry: the Lateralus package registry stores published crates. Each crate has: name, version, source code, and metadata.

Publishing workflow: lateralus-pkg publish uploads the crate. Pre-publish checks: compilation, tests, documentation, and license verification.

Version yanking: lateralus-pkg yank --version 1.2.3 marks a version as yanked. Yanked versions are not selected by the resolver but remain downloadable.

Crate ownership: crate owners can publish new versions and manage collaborators. Ownership transfer requires: current owner approval and registry administrator confirmation.

Download statistics: the registry tracks download counts per version. Statistics help: measure adoption, identify popular crates, and detect anomalies.

Security advisories: the registry supports publishing security advisories for crate versions. Advisories include: affected versions, severity, description, and recommended actions.

Alternative registries: organizations can host private registries for internal crates. Private registries use the same protocol as the public registry.

Registry mirroring: mirrors provide geographic distribution and redundancy. The build tool automatically selects the nearest mirror.

Crate documentation hosting: the registry hosts generated documentation for all published crates. Documentation is generated from the published source code.

Registry API: the registry provides a REST API for: crate search, version listing, dependency graph queries, and download. The API is used by the build tool and web interface.

6.3 Link-Time Optimization

LTO (Link-Time Optimization) performs optimization across module boundaries during linking. LTO improves: function inlining, dead code elimination, and constant propagation.

Thin LTO: a lightweight LTO variant that partitions the program into groups and optimizes within groups. Thin LTO provides most of LTO's benefit with lower memory usage.

Fat LTO: full LTO optimizes the entire program as a single unit. Fat LTO provides the best optimization but requires more memory and time.

LTO and separate compilation: LTO requires intermediate representation (IR) instead of machine

code. Each module produces an IR file that the linker optimizes.

LTO configuration: lateralus.toml specifies LTO mode: lto = 'off', lto = 'thin', lto = 'fat'. Default: off for debug, thin for release.

Cross-module inlining: LTO enables inlining functions defined in other modules. Cross-module inlining is the primary benefit of LTO for Lateralus programs.

Dead code elimination: LTO removes functions that are not reachable from the program entry point. DCE is more effective with LTO because it sees the entire program.

LTO and generics: monomorphized generic functions can be optimized across module boundaries with LTO. Without LTO, each module optimizes its own specializations independently.

LTO build time: thin LTO adds 10-20% to link time. Fat LTO can double link time for large programs. The trade-off is worthwhile for release builds.

LTO and debugging: LTO may rearrange and optimize code, making debugging harder. Use LTO only for release builds. Debug builds should use no LTO.

9.3 Module Testing Patterns

Unit test module: #[cfg(test)] mod tests { use super::*; #[test] fn test_function() { assert_eq!(1 + 1, 2); } }. Unit tests access private items in the parent module.

Integration tests: files in tests/ are separate crates that test the public API. Integration tests verify: API correctness, usage patterns, and error handling.

Test fixtures: shared test data and setup code in tests/common/mod.lat. Fixtures provide: test databases, mock services, and sample data.

Property-based testing: use the quickcheck crate to generate random inputs and verify properties. Property tests find edge cases that example-based tests miss.

Snapshot testing: capture function output and compare against saved snapshots. Snapshot tests detect unintended output changes. Update snapshots with --update-snapshots.

Mocking: use trait objects to inject mock implementations. Mock objects record calls and return predetermined values. Mocking tests component interactions.

Fuzzing: use the fuzz crate to generate random inputs and detect crashes. Fuzzing finds: buffer overflows, panics, and infinite loops. Fuzzing runs continuously.

Benchmark tests: use the criterion crate for statistical benchmarking. Benchmarks measure: throughput, latency, and memory usage. Results are compared across versions.

Coverage: use lateralus-cov to measure test coverage. Coverage reports show: line coverage, branch coverage, and function coverage. Target: 90%+ line coverage.

Test organization: group related tests in modules. Name tests descriptively: test_parse_valid_input,

test_parse_empty_input. Use test fixtures for shared setup.

2.3 Module Attributes

Module-level attributes apply to the entire module. #![allow(dead_code)] suppresses warnings for unused items in the module. #![deny(unsafe_code)] forbids unsafe blocks.

Inner attributes: #![attr] applies to the enclosing module. Outer attributes: #[attr] applies to the next item. Module-level attributes use inner syntax.

Module documentation: //! comments document the module itself. Module documentation appears at the top of generated docs. It describes the module's purpose and usage.

Feature gates: #![feature(name)] enables unstable features. Feature gates are only available on nightly. Stabilized features become available without gates.

Recursion limit: #![recursion_limit = '256'] sets the maximum macro expansion depth. The default is 128. Complex macros may need a higher limit.

Type length limit: #![type_length_limit = '1048576'] sets the maximum type name length. Deep generic nesting can produce very long type names.

No-std attribute: #![no_std] disables the standard library. The module uses only the core and alloc libraries. No-std is required for bare-metal and embedded development.

No-main attribute: #![no_main] disables the standard entry point. The program provides its own entry point using #[export_name = '_start']. Used for OS kernels and bootloaders.

Windows subsystem: #![windows_subsystem = 'windows'] creates a GUI application without a console window. Default is 'console' which creates a console window.

Crate-level documentation tests: documentation examples in module-level comments are compiled and run as tests. This ensures documentation stays accurate.

4.3 Extern Crate Declarations

Extern crate syntax: extern crate name; brings an external dependency into scope. In modern editions, extern crate is rarely needed because dependencies are automatically in scope.

Extern crate renaming: extern crate name as alias; renames the dependency. Renaming resolves: name conflicts between dependencies, and provides shorter names for frequently-used crates.

Sysroot crates: extern crate alloc; imports the allocation library. Sysroot crates (core, alloc, std) are always available without manifest declarations.

Proc-macro extern: extern crate proc_macro; imports the procedural macro library. This is required in proc-macro crates that define custom macros.

Extern crate and editions: in the 2018+ editions, extern crate is implicit for dependencies listed in the

manifest. Explicit extern crate is needed only for: sysroot crates and renaming.

Extern prelude: all dependencies listed in the manifest are available in the extern prelude. They can be referenced as: dependency_name::item without use declarations.

Extern crate in tests: test crates automatically depend on the crate being tested. extern crate my_crate; in tests imports the crate under test.

Extern crate macros: macros from external crates require: #[macro_use] extern crate name; (legacy) or use name::macro_name; (modern). Modern syntax is preferred.

Linking external crates: the linker resolves symbols from extern crates. Static linking includes the crate code in the binary. Dynamic linking loads the crate at runtime.

Crate versioning: multiple versions of the same crate can coexist in a dependency graph. Each crate sees only its declared version. Version deduplication removes redundant copies when possible.

7.3 Module Lifecycle Events

Module loading: when a module is first referenced, its code is loaded and initialized. Loading follows the dependency graph to ensure prerequisites are ready.

Module finalization: static destructors run when the program exits. Destructor order is the reverse of initialization order. Finalization is best-effort (not guaranteed on abnormal exit).

Module hot-reloading: development mode supports reloading changed modules without restarting. Hot reload preserves: global state, open connections, and UI state.

Module unloading: dynamically loaded modules can be unloaded when no longer needed. Unloading releases: code memory, static storage, and associated resources.

Module versioning: runtime module loading supports version negotiation. The loader selects the compatible version based on: API version, ABI version, and feature requirements.

Module metadata: each compiled module contains metadata: name, version, dependencies, feature flags, and compilation options. Metadata is accessible at runtime via std::module.

Module debugging: the debugger can list loaded modules, set breakpoints by module, and inspect module state. Module information includes: load address, size, and symbol table.

Module profiling: the profiler attributes CPU time and memory usage to modules. Module-level profiling identifies: hot modules, memory-heavy modules, and initialization bottlenecks.

Module isolation: sandboxed modules run with restricted capabilities. Isolated modules cannot: access the file system, network, or other modules' private state.

Module events: the runtime emits events for module lifecycle changes: loaded, initialized, unloaded, error. Event listeners can: log, monitor, and react to module changes.

5.4 Dependency Auditing

Supply chain security: dependencies can introduce vulnerabilities. Auditing verifies: known vulnerabilities, license compliance, and code integrity.

Vulnerability database: the RustSec Advisory Database tracks known vulnerabilities in published crates. The lateralus-audit tool checks dependencies against this database.

License scanning: lateralus-deny checks dependency licenses against an allow list. Incompatible licenses (GPL in MIT projects) are flagged. License compliance is enforced in CI.

Dependency pinning: the lock file pins exact versions. Updating dependencies is an explicit action. Pinning prevents: surprise breaking changes and untested versions.

Dependency minimization: regularly review and remove unused dependencies. Each dependency increases: build time, binary size, attack surface, and maintenance burden.

Dependency review: new dependencies should be reviewed for: code quality, maintenance activity, security history, and license compatibility. Review before adding to the manifest.

Supply chain attacks: malicious code in dependencies can: exfiltrate data, inject backdoors, and compromise builds. Mitigations: dependency review, pinning, auditing, and sandboxing.

Cargo-vet integration: lateralus-vet tracks which dependency versions have been audited. Audited versions are trusted. New versions require re-audit before use.

SBOM generation: Software Bill of Materials lists all dependencies with: name, version, license, and source. SBOMs are required for: government contracts, compliance, and security audits.

Dependency update policy: security updates are applied immediately. Feature updates are batched and tested. Major version updates require: migration planning, testing, and team review.

6.4 ABI Stability

The Lateralus ABI is not stable by default. The compiler is free to change: struct layout, calling conventions, and name mangling between versions.

Stable ABI: #[repr(C)] enforces the C ABI for struct layout. C ABI structs have: defined field order, alignment, and padding. C ABI enables: FFI compatibility and dynamic loading.

Calling conventions: extern 'C' fn() uses the C calling convention. extern 'Lateralus' fn() uses the default Lateralus convention. extern 'system' fn() uses the OS convention.

Name mangling: Lateralus uses a name mangling scheme to encode: module path, function name, generic parameters, and disambiguation. The mangling format is not stable.

Symbol versioning: shared libraries can expose multiple versions of a symbol. Callers bind to the version they were compiled against. New versions are added without breaking existing callers.

Dynamic library interface: shared libraries export a C-compatible interface. The interface includes:

function pointers, struct definitions, and version numbers.

ABI compatibility checking: the build tool can compare ABIs between versions. ABI breaks are detected and reported. ABI stability is enforced for libraries with stable ABI guarantees.

FFI bindings: the lateralus-bindgen tool generates Lateralus bindings from C/C++ headers. Bindings include: function signatures, struct definitions, and constant values.

Cbindgen: the cbindgen tool generates C/C++ headers from Lateralus source. Headers are generated for: #[no_mangle] functions and #[repr(C)] types.

Plugin ABI: the plugin system defines a stable ABI for dynamic plugins. Plugins are compiled as shared libraries. The host loads plugins at runtime via dlopen/LoadLibrary.

8.3 Cross-Compilation

Cross-compilation: compile for a different target architecture than the host. Example: compile on x86_64 Linux for aarch64 Linux.

Target triple: target specification format is arch-vendor-os-env. Examples: x86_64-unknown-linux-gnu, aarch64-apple-darwin, riscv64gc-unknown-none-elf.

Target installation: lateralus-target add riscv64gc-unknown-none-elf installs target support. Installation provides: standard library, linker scripts, and runtime support.

Sysroot: the cross-compilation sysroot contains: compiled standard library, platform headers, and link libraries for the target.

Cross-linker: cross-compilation requires a linker for the target. The linker is specified in: .cargo/config.toml as [target.triple].linker = 'cross-ld'.

Build script cross-compilation: build scripts run on the host, not the target. Build scripts use: host tools, host libraries, and host filesystem. They generate code for the target.

Conditional compilation and cross: cfg(target_arch) allows architecture-specific code. Platform abstractions use cfg to select the correct implementation for the target.

Testing cross-compiled code: cross-compiled tests can run on: emulators (QEMU), remote hardware, or containerized environments. The test runner is configured per target.

Embedded targets: embedded targets (thumbv7em-none-eabihf) have no standard library. Programs use: #![no_std], #![no_main], and direct hardware access.

Cross-compilation CI: CI pipelines cross-compile for all supported targets. Each target is built and tested. Cross-compilation ensures portability.

9.4 Module Documentation

Documentation generation: lateralus-doc generates HTML documentation from source code. Doc

comments (///) are extracted and rendered as Markdown.

Module-level docs: //! at the top of a file documents the module. Module docs describe: purpose, usage examples, and important notes.

Item-level docs: /// before an item documents that item. Item docs include: description, parameters, return value, errors, panics, examples, and safety.

Code examples in docs: examples in doc comments are compiled and run as tests. This ensures documentation accuracy. Examples use: fn main() { } or hidden fn main() { }.

Documentation links: [Type] creates links to other items. [module::Type] links to items in other modules. [crate::Type] links to items in the current crate.

Documentation search: generated documentation includes a search bar. Search indexes: item names, descriptions, and module paths. Search results are ranked by relevance.

Private documentation: lateralus-doc --document-private-items generates docs for private items. Private docs are used for: internal development, onboarding, and code review.

Documentation hosting: published crate documentation is hosted at docs.lateralus.dev. Documentation is generated automatically on publish. Each version has separate documentation.

Documentation badges: README files include badges linking to: documentation, crate page, build status, and coverage. Badges provide quick project health overview.

Documentation standards: all public items must have doc comments. Missing docs trigger warnings: #![warn(missing_docs)]. CI pipelines can enforce documentation completeness.

3.4 Privacy and Safety

Privacy enables safety: by hiding implementation details, modules prevent users from relying on internal invariants. Changes to private items do not break dependents.

Unsafe encapsulation: unsafe code is encapsulated in modules that provide safe public APIs. The module maintains invariants. Users cannot violate invariants through the safe API.

Sound abstractions: a module's API is sound if no sequence of safe operations can cause undefined behavior. Soundness is the primary safety property for module APIs.

Privacy and optimization: private fields enable the compiler to: change struct layout, inline accessors, and elide copies. Public fields constrain the compiler's optimization choices.

Privacy and evolution: private items can be changed, removed, or replaced without affecting dependents. Public items must be maintained for backward compatibility.

Semver and privacy: only public API changes affect semantic version. Adding private items is a patch change. Changing private implementations is a patch change.

Security boundaries: modules can enforce security invariants. A Validated[T] wrapper ensures data

has been validated. The constructor is the only way to create Validated values.

Trust boundaries: unsafe code trusts its own module's invariants. Other modules must uphold the public API contract. The trust boundary is the module boundary.

Audit scope: security audits focus on modules with unsafe code. Safe modules are audited for logic correctness. The module boundary reduces audit scope.

Privacy documentation: document why items are private. Document invariants that privacy protects. Document the safe API contract for modules with unsafe code.

10.1 Future Directions

Module versioning: future versions may support multiple API versions per module. Callers specify the API version they depend on. Migration tools assist with version transitions.

Hot module replacement: development environments will support hot-swapping module implementations. Running programs update modules without restart.

Module-level parallelism: future compilation pipelines will parallelize within-module compilation. Functions and types within a module compile concurrently.

Formal verification: modules may carry formal specifications (pre-conditions, post-conditions, invariants). The compiler or external tools verify specifications.

Capability-based modules: modules may declare required capabilities (file system, network, memory). The runtime enforces capabilities at the module boundary.

Module federation: distributed systems may share module definitions. Federated modules compile on different machines and link at deployment time.

AI-assisted module design: tools may suggest module boundaries based on code analysis. Suggestions consider: cohesion, coupling, and change frequency.

Module performance contracts: modules may declare performance requirements (latency, throughput). The compiler or runtime enforces performance contracts.

Module compatibility matrix: tools generate compatibility matrices showing which module versions work together. The matrix guides: dependency updates and migration planning.

Standardization: the module system specification may be standardized independently of the Lateralus language. Other languages could adopt the same module model.

2.4 Module Discovery

Automatic discovery: the compiler discovers modules by scanning the crate directory structure. Files ending in .lat are considered module source files.

Explicit declaration: mod name; in the parent module declares a child module. The compiler looks for

name.lat or name/mod.lat relative to the parent.

Module search paths: the compiler searches: the directory containing the parent module file, then the crate root. Additional search paths can be configured.

Orphan modules: files not declared by any parent module are orphans. The compiler warns about orphan files. Orphan modules are not compiled.

Module overrides: a manifest entry can override the default file location for a module. This allows: renaming modules, relocating modules, and generated modules.

Generated modules: build scripts can generate module source files. Generated modules are placed in the build output directory. The build system handles dependencies correctly.

Module inclusion: include!(filename) textually includes another file. Inclusion is different from module import. Included files share the including module's scope.

Path attributes: #[path = 'custom/path.lat'] overrides the default file location for a module. Path attributes enable non-standard directory layouts.

Auto-generated mod.lat: some tools automatically generate mod.lat files from directory contents. This reduces boilerplate for directories with many modules.

Module ignore patterns: .lateralusignore excludes files from module discovery. Ignore patterns follow glob syntax. Ignored files are not compiled or analyzed.

4.4 Advanced Import Patterns

Trait import: use module::Trait; imports the trait, making its methods available on types that implement it. Without the import, trait methods are not in scope.

Type alias import: use module::LongTypeName as Short; imports a type with a shorter alias. Aliases improve readability without affecting semantics.

Function import: use module::function; imports a free function. Imported functions are called without the module prefix: function() instead of module::function().

Constant import: use module::CONSTANT; imports a constant value. Imported constants are used directly: CONSTANT instead of module::CONSTANT.

Enum variant import: use module::Color::*; imports all variants of an enum. This allows: Red instead of Color::Red. Variant imports are convenient for pattern matching.

Macro import: use module::my_macro!; imports a macro. Macros have their own namespace. The ! suffix is required in the import declaration.

Selective re-export: pub use module::{Type1, Type2}; re-exports specific items. This curates the public API by choosing which items to expose.

Renaming re-export: pub use module::InternalName as PublicName; re-exports with a different

name. Renaming hides implementation naming conventions.

Conditional import: #[cfg(feature = 'advanced')] use module::advanced_feature; imports items conditionally. The import is only active when the condition is true.

Glob re-export with filter: there is no built-in syntax for glob re-export with exclusions. The pattern is: explicitly re-export desired items instead of using glob.

6.5 Build System Integration

Build system overview: the Lateralus build system orchestrates: dependency resolution, compilation, linking, testing, and packaging.

Build profiles: dev (fast compilation, debug info), release (optimization, no debug), test (test instrumentation), bench (optimization, benchmarking).

Custom profiles: [profile.custom] in the manifest defines custom build profiles. Custom profiles inherit from a base profile and override specific settings.

Build flags: lateralus-build --release compiles in release mode. --target specifies cross-compilation target. --features enables optional features.

Build cache: compiled artifacts are cached in the target/ directory. Switching profiles or features may trigger recompilation. The cache can be shared across projects.

Build environment variables: the build system sets: CARGO_MANIFEST_DIR, OUT_DIR, TARGET, HOST, PROFILE. Build scripts and procedural macros use these variables.

Build hooks: build.lat scripts run before compilation. They can: generate source code, compile C libraries, detect system configuration, and set build variables.

Build dependencies: build script dependencies are listed in [build-dependencies]. They are compiled for the host platform, not the target platform.

Build output: the build tool produces: executables (for binary crates), libraries (for library crates), documentation (with lateralus-doc), and test binaries.

Build reproducibility: reproducible builds produce identical binaries from the same source. Requirements: pinned dependencies, deterministic compilation, and no timestamps.

7.4 Module Runtime Representation

Module metadata struct: each compiled module contains a metadata section with: module name, version, dependency list, and initialization status.

Module symbol table: the symbol table maps: function names to addresses, type names to type descriptors, and constant names to values.

Module relocation table: relocations are applied when modules are loaded. Relocations resolve:

cross-module function calls, global variable references, and vtable entries.

Module debug information: DWARF debug info maps: source locations to machine code, variable names to registers/stack slots, and type definitions to memory layouts.

Module profiling data: instrumented modules contain: function entry/exit counters, branch probability counters, and loop trip counters.

Module code section: the .text section contains compiled machine code. Functions are laid out for cache efficiency. Hot functions are grouped together.

Module data section: the .data section contains: initialized global variables and static constants. The .bss section contains: zero-initialized globals.

Module read-only data: the .rodata section contains: string literals, constant tables, and vtables. Read-only data is shared between processes (copy-on-write).

Module thread-local storage: the .tdata and .tbss sections contain thread-local variables. Each thread gets its own copy of thread-local storage.

Module exception tables: the .eh_frame section contains unwinding information. Exception tables enable: panic unwinding, debugger stack traces, and profiler stack sampling.

8.4 Build Configuration Files

lateralus.toml: the crate manifest. Contains: [package] metadata, [dependencies], [dev-dependencies], [build-dependencies], [features], and [profile] sections.

lateralus.lock: the dependency lock file. Contains: exact versions, source URLs, and content hashes for all resolved dependencies. Committed to version control for binaries.

.lateralus/config.toml: user-level configuration. Contains: registry credentials, default features, linker paths, and build environment settings.

[workspace] configuration: members (list of workspace members), exclude (excluded directories), default-members (default build targets), resolver (dependency resolver version).

[package] fields: name, version, authors, edition, description, documentation, homepage, repository, license, keywords, categories, publish, and metadata.

[dependencies] syntax: name = 'version' (simple), name = { version = 'ver', features = ['f'], optional = true } (detailed). Dependencies support: registry, git, and path sources.

[features] definition: [features] section defines optional features. default = ['feature1'] enables default features. Features can enable: optional dependencies and other features.

[profile] optimization: [profile.release] opt-level = 3, lto = true, codegen-units = 1. [profile.dev] opt-level = 0, debug = true. Profiles control: optimization, debugging, and binary size.

Environment variables: LATERALUS_HOME sets the home directory. LATERALUS_TARGET_DIR

overrides the build output directory. LATERALUS_INCREMENTAL controls incremental compilation.

Build script outputs: build scripts communicate via stdout. println!('cargo:rustc-link-lib=name') links a library. println!('cargo:rustc-cfg=feature') sets a cfg condition.

3.5 Module Patterns Catalog

Service module: provides a public trait defining the service interface. Private struct implements the trait. Public factory function creates instances.

Repository module: encapsulates data access. Public interface: find, save, delete. Private implementation: SQL queries, connection management, error mapping.

Configuration module: reads and validates configuration. Public Config struct with public fields. Private parsing and validation functions.

Middleware module: wraps service calls with cross-cutting concerns. Public middleware function composes with the service. Private logging, timing, and error handling.

Plugin module: defines a plugin interface (trait) and plugin registry. Plugins register implementations. The registry dispatches to registered plugins.

Event module: defines event types and an event bus. Publishers emit events. Subscribers register handlers. The bus dispatches events to matching handlers.

Error module: defines error types for the crate. Public error enum with variants for each error case. Implements Display, Debug, and From conversions.

Prelude module: re-exports commonly used items from the crate. Users import with: use crate::prelude::*;. The prelude simplifies imports for typical usage.

Internal module: private module containing shared implementation details. Sibling public modules depend on the internal module. The internal module is not exposed in the API.

Test utilities module: #[cfg(test)] pub mod test_utils provides: mock objects, test fixtures, assertion helpers, and test data generators for use by multiple test modules.

9.5 Module Security Patterns

Capability tokens: a module issues opaque tokens representing capabilities. Operations require passing the token. The token cannot be forged because the type is private.

Tainted types: input from external sources is wrapped in Tainted[T]. Processing requires explicit validation. The Tainted wrapper prevents using unvalidated data.

Privilege separation: high-privilege operations are in a separate module. The module's API is minimal and auditable. Low-privilege code calls the high-privilege API.

Sandboxed execution: modules can run user code in a sandboxed environment. The sandbox

restricts: memory allocation, system calls, and execution time.

Cryptographic module: cryptographic operations are encapsulated in a dedicated module. Key material is stored in private fields. The API prevents: key leakage, misuse, and timing attacks.

Input validation module: all external input passes through a validation module. The module rejects: malformed data, oversized inputs, and injection attempts.

Audit logging module: security-relevant operations are logged. The logging module is tamper-resistant. Log entries include: timestamp, actor, action, and result.

Secret management module: secrets (API keys, passwords) are stored in a dedicated module. The module provides: secure storage, rotation, and access control.

Rate limiting module: the module tracks request rates per client. Exceeding the rate limit returns an error. The module prevents: denial of service and brute force attacks.

Access control module: the module enforces authorization policies. Policies define: who can access what resources with what permissions. The module is the single enforcement point.

10.2 Comparison with Other Module Systems

ML modules: ML uses functors (parameterized modules). Lateralus does not have functors. Generics and traits provide similar abstraction power.

Haskell modules: Haskell modules use export lists. Lateralus uses pub/private visibility. Both support qualified imports and re-exports.

Java packages: Java packages use directory structure. Lateralus modules also use directory structure. Java lacks: fine-grained visibility (only public/protected/private).

Python modules: Python modules are files. Import is dynamic (at runtime). Lateralus imports are static (at compile time). Static imports enable: compile-time checking and optimization.

C++ modules (C++20): C++ modules replace headers. Lateralus modules are similar in concept. C++ modules support: import declarations, module partitions, and header unit imports.

Go packages: Go packages use directory-based organization. All files in a directory belong to the same package. Go lacks: fine-grained visibility (only exported/unexported).

Rust modules: Lateralus modules are directly inspired by Rust modules. Key similarities: file-based modules, visibility modifiers, use declarations, and crate organization.

JavaScript modules (ESM): ESM uses export/import syntax. ESM is dynamic (imports can be computed). Lateralus imports are static. ESM supports: default exports and namespace imports.

Swift modules: Swift modules correspond to frameworks/packages. Access control: open, public, internal, fileprivate, private. Swift has more visibility levels than Lateralus.

Zig modules: Zig uses file-based modules similar to Lateralus. Zig modules are comptime-resolved.

Zig lacks: a package registry (uses direct URL dependencies).

4.5 Import Performance

Import cost: each import adds a lookup entry. Large numbers of glob imports increase name resolution time. Prefer explicit imports for large projects.

Unused import detection: the compiler detects unused imports and emits warnings. Removing unused imports improves compilation speed and code clarity.

Import deduplication: if the same item is imported through multiple paths, the compiler deduplicates. Only one copy of the item exists in the symbol table.

Import parallelism: import resolution can be parallelized across independent modules. The compiler processes imports concurrently when modules do not depend on each other.

Import caching: resolved import paths are cached. Incremental compilation reuses cached imports when source files have not changed.

Import ordering effects: import order does not affect semantics. The compiler processes all imports before resolving names. Import order is a style choice.

Import compile time: large import trees (many transitive dependencies) increase compile time. Techniques to reduce: minimize dependency count, use feature flags, and split large crates.

Import binary size: unused imports do not affect binary size. Dead code elimination removes unreferenced items. Only actually-used items contribute to the final binary.

Import and IDE: IDEs use import information for: auto-completion, go-to-definition, and refactoring. Fast import resolution enables responsive IDE features.

Auto-import: IDEs can automatically add import declarations when a user types an unresolved name. Auto-import searches: the current crate, dependencies, and the standard library.

5.5 Dependency Graph Visualization

Graph generation: lateralus-deps generates a dependency graph in DOT format. The graph shows: crates as nodes, dependencies as edges, and features as labels.

Graph filtering: lateralus-deps --filter name shows only the subgraph rooted at the specified crate. Filtering focuses on a specific dependency and its transitive dependencies.

Duplicate detection: the graph tool highlights crates that appear in multiple versions. Duplicates increase binary size and may cause type mismatches.

Tree view: lateralus-deps --tree displays dependencies as an indented tree. The tree shows: direct dependencies, transitive dependencies, and their versions.

Reverse dependencies: lateralus-deps --reverse name shows which crates depend on the specified

crate. Reverse dependencies help assess the impact of changes.

Graph metrics: the tool reports: total crate count, maximum dependency depth, duplicate count, and feature flag count. Metrics guide dependency optimization.

Circular dependency detection: the tool detects cycles in the dependency graph. Cycles are errors that prevent compilation. The tool shows the cycle path.

Build time analysis: the tool annotates nodes with compilation time. Long-compilation dependencies are highlighted. Build time analysis guides optimization.

Size analysis: lateralus-deps --size reports the size contribution of each dependency. Large dependencies are highlighted. Size analysis guides binary size optimization.

Security analysis: the tool flags dependencies with known vulnerabilities. Vulnerable dependencies are highlighted in red. The tool links to advisory details.

7.5 Dynamic Module Loading

Dynamic loading API: std::module::load(path) loads a shared library at runtime. The function returns a Module handle. The handle provides access to the module's symbols.

Symbol lookup: module.get::<fn()>(name) looks up a function by name. module.get::<*const T>(name) looks up a global variable. Type safety is the caller's responsibility.

Plugin interface: plugins implement a defined trait. The host loads the plugin library and calls a factory function. The factory returns a trait object.

Module lifetime: loaded modules remain in memory until explicitly unloaded or the program exits. Module handles are reference-counted. Unloading occurs when the count reaches zero.

ABI compatibility: dynamic modules must match the host's ABI. Version mismatch is detected at load time. The loader checks: ABI version, compiler version, and target triple.

Error handling: load failures return descriptive errors: file not found, symbol not found, ABI mismatch, initialization failure. Errors include the module path and reason.

Security: loaded modules run with full program privileges. Sandboxing is optional. Untrusted modules should be loaded in a restricted environment.

Hot reload: development mode supports reloading changed modules. The host detects file changes, unloads the old module, and loads the new one. State is preserved via shared memory.

Cross-platform loading: the API abstracts platform differences. Linux uses dlopen/dlsym, macOS uses dyld, Windows uses LoadLibrary/GetProcAddress. The API is uniform.

Module search paths: the loader searches: the application directory, system library paths, and configured search paths. Search order is platform-dependent.

8.5 Feature Flag Patterns

Additive features: features only add functionality, never remove it. A crate compiled with feature A and a crate compiled without feature A can coexist.

Default features: default = ['std'] enables the std feature by default. Users disable defaults with: default-features = false. Default features provide common functionality.

Feature groups: related features are grouped: serialization = ['serde', 'json', 'yaml']. Enabling the group enables all member features.

Platform features: features that enable platform-specific code. Example: feature 'linux-io-uring' enables io_uring support on Linux. Platform features are not default.

Unstable features: features prefixed with 'unstable-' indicate experimental functionality. Unstable features may change or be removed. They are not covered by semver.

Feature documentation: each feature is documented in the manifest and README. Documentation describes: what the feature enables, performance implications, and dependencies added.

Feature testing: CI tests all feature combinations. The lateralus-hack tool generates all combinations. Testing ensures features compose correctly.

Feature impact: each feature's impact is documented: binary size increase, compilation time increase, and new dependencies. Impact data guides feature selection.

Compile-time feature detection: #[cfg(feature = 'name')] enables conditional compilation. The cfg attribute works in: modules, functions, structs, and expressions.

Feature-gated APIs: public items can be feature-gated. Feature-gated items appear in documentation with a feature badge. Users know which features are needed.

6.6 Compilation Pipeline

Pipeline stages: source parsing, macro expansion, name resolution, type checking, borrow checking, MIR generation, optimization, code generation, and linking.

Parsing: the parser converts source text to an abstract syntax tree (AST). Parsing is per-file and parallelizable. Parse errors are recovered to continue analysis.

Macro expansion: macros are expanded in the AST. Expansion may introduce new items. Name resolution re-runs after expansion. Multiple expansion rounds may occur.

Type checking: the type checker verifies type correctness. It resolves: method calls, operator overloading, trait implementations, and generic instantiations.

Borrow checking: the borrow checker verifies memory safety. It ensures: no use-after-free, no data races, and correct lifetime relationships. Borrow checking is per-function.

MIR generation: the compiler lowers the typed AST to Mid-level Intermediate Representation. MIR is

a control-flow graph suitable for analysis and optimization.

Optimization: MIR optimizations include: constant propagation, dead code elimination, inlining, loop unrolling, and devirtualization. Optimization levels control the extent.

Code generation: the compiler lowers MIR to machine code via LLVM. LLVM performs additional optimizations: instruction selection, register allocation, and scheduling.

Linking: the linker combines object files, resolves symbols, and produces the final executable or library. LTO occurs during linking.

Pipeline parallelism: independent stages can overlap. Parsing module B while type-checking module A. Pipeline parallelism reduces total build time.

9.6 Module Migration Guide

Splitting a large module: identify cohesive groups of items. Move each group to a new submodule. Re-export public items from the parent. Update imports in dependents.

Merging small modules: combine modules with high coupling. Move items into the target module. Remove the source module. Update imports in dependents.

Renaming a module: create the new module with the desired name. Add pub use new_module as old_module; for backward compatibility. Deprecate the old name. Remove after migration period.

Extracting a crate: move the module to a new crate. Add the new crate as a dependency of the original. Re-export from the original for backward compatibility. Remove re-exports after migration.

Visibility changes: narrowing visibility (pub to pub(crate)) may break dependents. Use a deprecation period. Document the change in the changelog.

API changes: use #[deprecated] to mark old APIs. Provide migration documentation. Maintain compatibility for one major version. Remove deprecated items in the next major version.

Feature migration: when replacing a feature flag, add the new feature, deprecate the old feature (which enables the new one), and remove the old feature in the next major version.

Dependency migration: when replacing a dependency, add the new dependency, update internal code, re-export the new API, deprecate the old re-exports, and remove in the next version.

Automated migration: lateralus-fix applies automatic code transformations. Fix rules handle: import path changes, renamed items, and deprecated API replacements.

Migration testing: run the full test suite after each migration step. Test both the old and new APIs during the transition period. Remove old API tests after the migration completes.

A.1 Module System Grammar

module-declaration: 'mod' IDENT ';' | 'mod' IDENT '{' module-body '}'. Module declarations introduce

a new module in the current scope.

use-declaration: 'use' use-tree ';'. Use declarations bring items into scope from other modules.

use-tree: simple-path | simple-path '::' '*' | simple-path '::' '{' use-tree-list '}' | simple-path 'as' IDENT.

visibility: 'pub' | 'pub' '(' 'crate' ')' | 'pub' '(' 'super' ')' | 'pub' '(' 'in' simple-path ')'. Visibility modifiers control item accessibility.

extern-crate: 'extern' 'crate' IDENT ';' | 'extern' 'crate' IDENT 'as' IDENT ';'. Extern crate declarations bring external dependencies into scope.

module-body: inner-attribute* item*. The module body consists of inner attributes followed by item definitions.

item: visibility? (function | struct | enum | trait | impl | type-alias | const | static | module-declaration | use-declaration | extern-crate). Items are the building blocks of modules.

simple-path: '::'? path-segment ('::' path-segment)*. Simple paths identify modules and items.

path-segment: IDENT | 'crate' | 'self' | 'super'. Path segments navigate the module hierarchy.

re-export: 'pub' use-declaration. Re-exports make imported items available to users of the current module.

cfg-module: '#[cfg(' cfg-predicate ')]' module-declaration. Conditional module declarations include modules only when the predicate is satisfied.

inline-module: 'mod' IDENT '{' inner-attribute* item* '}'. Inline modules define a module within the current file without a separate source file.

A.2 Module System Error Catalog

E0001: Module not found. The compiler cannot locate the source file for a declared module. Check file naming and directory structure.

E0002: Circular module dependency. Module A depends on B and B depends on A. Break the cycle by extracting shared code into a third module.

E0003: Private item access. An item is not visible from the accessing module. Change visibility or access through a public API.

E0004: Ambiguous import. Two glob imports provide items with the same name. Resolve by using explicit imports or qualified paths.

E0005: Duplicate item definition. Two items with the same name exist in the same scope. Rename one item or move it to a different module.

E0006: Unresolved import. The imported path does not resolve to an item. Check the path, module visibility, and dependency declarations.

E0007: Unused import. An imported item is not used in the module. Remove the unused import to clean up the code.

E0008: Orphan rule violation. A trait implementation violates the orphan rule. Either the trait or the type must be defined in the current crate.

E0009: Feature not enabled. An item requires a feature flag that is not enabled. Add the feature to the dependency declaration or enable it in the manifest.

E0010: ABI mismatch. A dynamically loaded module was compiled with a different ABI version. Recompile the module with the matching compiler version.

E0011: Deprecated item usage. The code uses a deprecated item. Migrate to the recommended replacement as described in the deprecation notice.

E0012: Module initialization failure. A static initializer panicked during module initialization. Simplify the initializer or use lazy initialization.

A.3 Module System Quick Reference

Declare a module: mod name; (file-based) or mod name { ... } (inline). File modules use name.lat or name/mod.lat.

Import an item: use path::item; makes item available without prefix. use path::item as alias; creates an alias.

Import multiple: use path::{item1, item2}; imports multiple items. use path::*; imports all public items.

Re-export: pub use path::item; makes a submodule's item part of the current module's public API.

Visibility: pub (public everywhere), pub(crate) (crate-only), pub(super) (parent module only), pub(in path) (specific module).

External dependency: add to [dependencies] in lateralus.toml. Access as: dependency_name::item.

Conditional module: #[cfg(target_os = 'linux')] mod platform; includes module only on Linux.

Test module: #[cfg(test)] mod tests { use super::*; #[test] fn test() { ... } }.

Module documentation: //! comment at the top of the file documents the module. /// comment documents the next item.

Module path: crate::module::item (from root), self::item (current module), super::item (parent module).

Feature-gated module: #[cfg(feature = 'advanced')] mod advanced; includes module only when the feature is enabled.

Workspace: [workspace] in root lateralus.toml. members = ['crate1', 'crate2']. Shared build output and lock file.

References

[1] Cardelli, L. Program Fragments, Linking, and Modularization. POPL 1997.

[2] Harper, R. and Stone, C. A Type-Theoretic Interpretation of Standard ML. MIT Press, 2000.

[3] Leroy, X. A Modular Module System. JFP, 2000.

[4] Syme, D. et al. The F# Component Design Guidelines. Microsoft, 2010.

[5] The Rust Reference: Modules. The Rust Project Developers, 2024.