Lateralus Language Specification v3.0

bad-antics | January 2025 | Language Specification

Abstract

This document is the authoritative specification for version 3.0 of the Lateralus programming language. It covers lexical structure, the type system, expressions, statements, control flow, ownership and borrowing, pattern matching, traits, generics, concurrency, unsafe code, the standard library interface, and language evolution.

1 Introduction

Lateralus version 3.0 represents a major evolution of the language specification, introducing refined ownership semantics, an expanded type system, improved pattern matching, and a formalized module system. This document serves as the authoritative reference for the Lateralus 3.0 language.

Lateralus is a systems programming language designed for safety, performance, and expressiveness. The language combines: algebraic data types, ownership-based memory management, pipeline-native computation, and zero-cost abstractions.

This specification covers: lexical structure, types, expressions, statements, items, ownership and borrowing, pattern matching, modules, traits, generics, concurrency, unsafe code, and the standard library interface.

2 Lexical Structure

Source files are encoded in UTF-8. The lexer processes: whitespace, comments, identifiers, keywords, literals, operators, and delimiters.

Comments: // line comments extend to the end of the line. /* block comments */ can nest. /// doc comments document the following item. //! inner doc comments document the enclosing item.

Identifiers: start with a letter or underscore, followed by letters, digits, or underscores. Raw identifiers r#name allow keywords as identifiers.

Keywords: fn, let, mut, const, static, struct, enum, trait, impl, type, mod, use, pub, crate, self, super, as, in, for, while, loop, if, else, match, return, break, continue, move, ref, unsafe, async, await, dyn, where.

Integer literals: decimal (42), hexadecimal (0xFF), octal (0o77), binary (0b1010). Type suffixes: 42i32, 42u64, 42usize. Underscores for readability: 1_000_000.

Float literals: 3.14, 3.14f64, 1e10, 2.5E-3. Float types: f32 (32-bit IEEE 754), f64 (64-bit IEEE 754). No implicit float-to-int conversion.

String literals: double-quoted ("hello"), raw strings (r"no escapes"), byte strings (b"bytes"). Escape sequences: \n, \t, \r, \\, \0, \xNN, \u{NNNN}.

Character literals: single-quoted ('a', '\n', '\u{1F600}'). Characters are Unicode scalar values (32-bit).

3 Type System

Primitive types: bool, char, i8/i16/i32/i64/i128/isize, u8/u16/u32/u64/u128/usize, f32/f64, str (string slice), () (unit type), ! (never type).

Compound types: tuples (T1, T2, ...), arrays [T; N] (fixed-size), slices [T] (dynamically-sized), references &T and &mut T.

User-defined types: structs (named fields, tuple structs, unit structs), enums (variants with optional data), and type aliases.

Generic types: struct Vec[T], enum Option[T], fn map[T, U](f: fn(T) -> U). Type parameters are monomorphized at compile time.

Trait objects: dyn Trait represents a dynamically-dispatched trait. Trait objects use: vtable pointers, fat references (&dyn Trait), and boxed trait objects (Box[dyn Trait]).

Lifetime annotations: &'a T indicates a reference with lifetime 'a. Lifetimes ensure references are valid. The compiler infers lifetimes when possible.

The never type (!): represents computations that never complete. Functions returning ! never return normally. Used for: panic, infinite loops, and process exit.

4 Expressions

Literal expressions: integer (42), float (3.14), boolean (true, false), character ('a'), string ("hello"), array ([1, 2, 3]), tuple ((1, 'a')).

Path expressions: crate::module::item, self::item, super::item. Paths resolve to: functions, constants, types, and modules.

Block expressions: { statements; expression }. The block evaluates to its final expression. Blocks introduce a new scope. Variables declared in the block are not accessible outside.

Operator expressions: arithmetic (+, -, *, /, %%), comparison (==, !=, <, >, <=, >=), logical (&&, ||, !), bitwise (&, |, ^, <<, >>).

Assignment expressions: x = value (simple), x += value (compound). Compound assignment operators: +=, -=, *=, /=, %%=, &=, |=, ^=, <<=, >>=.

Call expressions: function(args), method.call(args), Type::associated(args). Function calls are expressions that evaluate to the return value.

Field access: value.field, value.0 (tuple field). Field access resolves at compile time. Invalid field names produce compile errors.

Index expressions: array[index], slice[index]. Index out of bounds panics at runtime (debug mode) or is undefined behavior (unchecked).

Pipeline expressions: value |> function applies function to value. Pipelines chain transformations: data |> parse |> validate |> transform |> output.

5 Statements

Let statements: let x = value; (immutable binding), let mut x = value; (mutable binding), let x: Type = value; (explicit type).

Expression statements: expression; evaluates the expression and discards the result. Used for: function calls with side effects, assignments, and method calls.

Item statements: fn, struct, enum, trait, impl, mod, use, const, static declarations within blocks are item statements. Item statements are hoisted to the block's scope.

6 Control Flow

If expressions: if condition { then } else { otherwise }. If is an expression: let x = if cond { a } else { b };. Else-if chains: if c1 { } else if c2 { } else { }.

Loop expressions: loop { body } runs indefinitely. break exits the loop. break value exits with a value. continue skips to the next iteration.

While loops: while condition { body }. While-let: while let Some(x) = iter.next() { use(x) }. While loops cannot produce values (evaluate to ()).

For loops: for item in iterable { body }. For loops desugar to: IntoIterator::into_iter(iterable).for_each(|item| { body }). For loops support pattern binding.

Match expressions: match value { pattern => result, ... }. Match is exhaustive: all possible values must be covered. The compiler verifies exhaustiveness.

7 Ownership and Borrowing

Ownership rules: each value has exactly one owner. When the owner goes out of scope, the value is dropped. Ownership can be transferred (moved) to another binding.

Move semantics: let y = x; moves x to y. After the move, x is no longer valid. Move semantics apply to types that do not implement Copy.

Copy semantics: types implementing Copy are duplicated on assignment. Copy types include: integers, floats, booleans, characters, and tuples of Copy types.

Borrowing: &x creates an immutable reference. &mut x creates a mutable reference. Rules: any

number of immutable references OR exactly one mutable reference. No reference outlives the referent.

Lifetime elision: the compiler infers lifetimes in common patterns. Elision rules: each input reference gets a distinct lifetime; if there is one input lifetime, it is assigned to all outputs; if self is a parameter, its lifetime is assigned to outputs.

Drop semantics: when a value goes out of scope, its Drop::drop method is called. Drop order: variables are dropped in reverse declaration order. Struct fields are dropped in declaration order.

8 Pattern Matching

Literal patterns: match x { 0 => 'zero', 1 => 'one', _ => 'other' }. Literal patterns match exact values. The wildcard _ matches any value.

Binding patterns: match x { n => use(n) }. The variable n is bound to the matched value. Bindings introduce new variables in the match arm's scope.

Struct patterns: match point { Point { x, y } => use(x, y) }. Struct patterns destructure struct fields. Fields can be renamed: Point { x: a, y: b }.

Enum patterns: match opt { Some(v) => use(v), None => default() }. Enum patterns match specific variants and bind variant data.

Reference patterns: match &x { &val => use(val) }. Reference patterns dereference the matched value. Ref patterns: match x { ref r => use(r) } borrow the value.

Guard patterns: match x { n if n > 0 => positive(), _ => non_positive() }. Guards add conditions to patterns. Guards are evaluated after the pattern matches.

Or patterns: match x { 1 | 2 | 3 => small(), _ => large() }. Or patterns match any of the alternatives. All alternatives must bind the same variables.

9 Traits

Trait definition: trait Name { fn method(&self) -> Type; fn default_method(&self) { default_body } }. Traits define shared behavior through method signatures.

Trait implementation: impl Trait for Type { fn method(&self) -> Type { body } }. Types can implement multiple traits. Trait implementations must cover all required methods.

Associated types: trait Iterator { type Item; fn next(&mut self) -> Option[Self::Item]; }. Associated types are specified by the implementor.

Trait bounds: fn process[T: Display + Clone](value: T). Trait bounds constrain generic parameters. Where clauses: fn f[T](x: T) where T: Display + Clone.

Supertraits: trait Printable: Display + Debug { }. Implementing Printable requires implementing

Display and Debug. Supertraits define prerequisite behavior.

Object safety: a trait is object-safe if all methods have a receiver (&self, &mut self, or self) and no generic methods. Object-safe traits can be used as trait objects.

10 Generics

Generic functions: fn max[T: Ord](a: T, b: T) -> T { if a >= b { a } else { b } }. The compiler generates specialized code for each concrete type used.

Generic structs: struct Pair[T, U] { first: T, second: U }. Generic structs are parameterized by type. Instantiation: Pair { first: 1, second: 'a' }.

Generic enums: enum Result[T, E] { Ok(T), Err(E) }. Generic enums are the foundation of error handling and optional values in Lateralus.

Const generics: struct Array[T, const N: usize] { data: [T; N] }. Const generics allow types parameterized by constant values. Used for: fixed-size arrays, SIMD lanes.

Monomorphization: the compiler generates a separate copy of each generic function for each concrete type. Monomorphization produces fast code but increases binary size.

11 Concurrency

Threads: std::thread::spawn(|| { body }) creates a new OS thread. Threads share the process address space. Thread::join() waits for completion.

Send and Sync: Send types can be transferred between threads. Sync types can be shared between threads (via &T). The compiler enforces Send and Sync at compile time.

Mutex: Mutex[T] provides mutual exclusion. lock() returns a guard that dereferences to &mut T. The guard releases the lock on drop. Poisoned mutexes indicate panics.

Channels: std::sync::mpsc provides multiple-producer, single-consumer channels. tx.send(value) sends. rx.recv() receives. Channels are typed and thread-safe.

Async/await: async fn process() -> Result { let data = fetch().await?; transform(data) }. Async functions return futures. .await suspends until the future completes.

Async runtime: the async runtime schedules futures on a thread pool. Popular runtimes: tokio (multi-threaded), smol (lightweight). The runtime is not part of the language.

12 Unsafe Code

Unsafe blocks: unsafe { raw_pointer.read() }. Unsafe blocks allow: raw pointer dereference, calling unsafe functions, accessing mutable statics, and implementing unsafe traits.

Raw pointers: *const T (immutable) and *mut T (mutable). Raw pointers can be null, dangling, or unaligned. Dereferencing requires unsafe.

Unsafe functions: unsafe fn dangerous() { }. Callers must use unsafe blocks. Unsafe functions document: preconditions, invariants, and undefined behavior conditions.

FFI: extern 'C' fn callback(x: i32) -> i32. Foreign function interface uses C calling convention. extern blocks declare foreign functions.

13 Standard Library

Collections: Vec[T] (growable array), HashMap[K, V] (hash map), BTreeMap[K, V] (sorted map), HashSet[T], BTreeSet[T], VecDeque[T] (double-ended queue), LinkedList[T].

String types: String (owned, growable), &str (borrowed slice), OsString (OS-native), CString (C-compatible). String operations: concatenation, slicing, searching, formatting.

I/O: std::io provides Read, Write, BufRead, Seek traits. File, stdin, stdout, stderr implement I/O traits. Buffered I/O: BufReader, BufWriter.

Networking: std::net provides TcpStream, TcpListener, UdpSocket. Address types: SocketAddr, IpAddr (v4 and v6). Connection management and timeout configuration.

Error handling: std::error::Error trait. Box[dyn Error] for type-erased errors. The ? operator propagates errors. From trait enables error conversion.

14 Conclusion

Lateralus 3.0 provides a complete systems programming language with: memory safety through ownership, expressive types, powerful pattern matching, and pipeline-native computation. This specification defines the language precisely for implementors and users.

2.1 Operator Tokens

Arithmetic operators: + (add), - (subtract/negate), * (multiply), / (divide), %% (remainder). Operator precedence follows standard mathematical convention.

Comparison operators: == (equal), != (not equal), < (less than), > (greater than), <= (less or equal), >= (greater or equal). Comparisons return bool.

Logical operators: && (short-circuit and), || (short-circuit or), ! (not). Logical operators work on bool values only.

Bitwise operators: & (and), | (or), ^ (xor), ! (not), << (left shift), >> (right shift). Bitwise operators work on integer types.

Assignment operators: = (assign), += -= *= /= %%= (compound arithmetic), &= |= ^= <<= >>=

(compound bitwise). Compound assignment is syntactic sugar.

Range operators: .. (exclusive range), ..= (inclusive range), ..expr (range to), expr.. (range from). Ranges implement Iterator.

Reference operators: & (borrow), &mut (mutable borrow), * (dereference). Reference operators interact with the ownership system.

Pipeline operator: |> (pipe). Left operand becomes the first argument of the right operand. Pipelines are left-associative.

Error propagation: ? (try operator). expr? is sugar for: match expr { Ok(v) => v, Err(e) => return Err(e.into()) }. The ? operator works in functions returning Result.

Type cast: as (type cast). Numeric casts: 42u8 as u32. Pointer casts: &x as *const i32. Casts can lose precision or change representation.

3.1 Numeric Types Detail

Signed integers: i8 (-128 to 127), i16 (-32768 to 32767), i32 (-2^31 to 2^31-1), i64 (-2^63 to 2^63-1), i128 (-2^127 to 2^127-1), isize (pointer-sized).

Unsigned integers: u8 (0 to 255), u16 (0 to 65535), u32 (0 to 2^32-1), u64 (0 to 2^64-1), u128 (0 to 2^128-1), usize (pointer-sized).

Integer overflow: in debug mode, overflow panics. In release mode, overflow wraps (two's complement). Explicit wrapping: wrapping_add(), checked_add(), saturating_add(), overflowing_add().

Floating-point: f32 (IEEE 754 single, 6-7 significant digits), f64 (IEEE 754 double, 15-16 significant digits). NaN, infinity, and negative zero are supported.

Floating-point operations: arithmetic follows IEEE 754 rules. NaN propagation: any operation with NaN produces NaN. NaN != NaN (NaN is not equal to itself).

Numeric conversions: from i32 to i64 is lossless (as keyword). From i64 to i32 may truncate. From f64 to i32 rounds toward zero. From i32 to f32 may lose precision.

Platform-dependent types: isize and usize have platform-dependent size (32 or 64 bits). Used for: array indexing, pointer arithmetic, and collection sizes.

Numeric traits: Add, Sub, Mul, Div, Rem for arithmetic. Neg for negation. Not, BitAnd, BitOr, BitXor, Shl, Shr for bitwise. PartialOrd, Ord for comparison.

Numeric literals: type suffix determines the type: 42i32, 42u64, 3.14f32. Without suffix: integer literals default to i32, float literals to f64.

Numeric formatting: Display formats for humans. Debug formats for debugging. LowerHex, UpperHex, Octal, Binary for base conversions. Precision specifiers for floats.

4.1 Closure Expressions

Closure syntax: |params| body, |params| -> Type { body }. Closures capture variables from the enclosing scope. Closures are anonymous functions.

Capture modes: by reference (Fn), by mutable reference (FnMut), by value (FnOnce). The compiler infers the least restrictive capture mode.

Move closures: move |params| body forces capture by value. Move closures take ownership of captured variables. Required for closures sent to other threads.

Closure types: each closure has a unique anonymous type. Closures implement Fn, FnMut, or FnOnce traits. Function pointers (fn) are a subset of closures.

Closure return types: closures infer return types from the body. Explicit return types can be specified: |x: i32| -> i32 { x + 1 }.

Higher-order functions: functions that accept closures: fn map[T,U](v: Vec[T], f: impl Fn(T) -> U) -> Vec[U]. Higher-order functions enable functional programming patterns.

Closure performance: closures are compiled to struct types with captured variables as fields. Closure calls are inlined when possible. Zero-cost abstraction.

Closure limitations: closures cannot be recursive directly (no name to reference). Workaround: use a named function or a trait object.

Closure and lifetime: closures that borrow must not outlive the borrowed values. The compiler enforces lifetime constraints on closure captures.

Closure coercion: non-capturing closures coerce to function pointers. |x: i32| x + 1 coerces to fn(i32) -> i32. This enables: C FFI callbacks.

5.1 Declaration Statements

Const declarations: const NAME: Type = value; evaluated at compile time. Constants are inlined at each usage site. Constants must have a type annotation.

Static declarations: static NAME: Type = value; allocated for the program's lifetime. Statics have a fixed memory address. static mut requires unsafe to access.

Type alias declarations: type Name = ExistingType; creates a new name for an existing type. Type aliases do not create new types; they are transparent.

Extern declarations: extern 'C' { fn foreign_function(x: i32) -> i32; }. Extern blocks declare functions with foreign calling conventions.

Attribute declarations: #[attribute] items. Attributes provide metadata. Built-in attributes: #[derive(...)], #[cfg(...)], #[test], #[inline], #[allow(...)], #[deny(...)].

Macro declarations: macro_rules! name { (pattern) => { expansion }; }. Declarative macros transform

syntax. Macros are expanded before compilation.

Use declarations: use path::item; imports items into scope. use path::*; glob import. use path::item as alias; renaming import.

Mod declarations: mod name; declares a module (file-based). mod name { items } defines an inline module.

Impl declarations: impl Type { methods } adds methods to a type. impl Trait for Type { methods } implements a trait.

Trait declarations: trait Name { methods } defines a trait. Traits can have: required methods, default methods, associated types, and associated constants.

6.1 Advanced Control Flow

Labeled blocks: 'label: { body }. break 'label exits the labeled block. Labels allow breaking out of nested constructs.

Labeled loops: 'outer: loop { 'inner: loop { break 'outer; } }. break 'label exits the specified loop. continue 'label skips to the next iteration of the specified loop.

If-let expressions: if let Some(x) = option { use(x) } else { default() }. If-let simplifies matching a single pattern.

While-let loops: while let Some(x) = iter.next() { process(x) }. While-let loops until the pattern fails to match.

Let-else: let Some(x) = option else { return None; };. Let-else provides an early return when a pattern does not match.

Match ergonomics: match &option { Some(x) => {} None => {} }. The compiler automatically adds reference patterns. Match ergonomics reduce boilerplate.

Exhaustive matching: the compiler verifies that match expressions cover all possible values. Missing patterns are compile errors. Wildcards (_) catch remaining patterns.

Match arm expressions: each match arm is an expression. All arms must produce compatible types. Match expressions evaluate to the selected arm's expression.

Diverging expressions: expressions of type ! (never) are compatible with any type. return, break, continue, panic!(), and loop {} are diverging.

Short-circuit evaluation: && and || evaluate the right operand only if necessary. && returns false if the left is false. || returns true if the left is true.

7.1 Ownership Advanced Topics

Partial moves: let (a, b) = tuple; moves fields independently. After a partial move, the original tuple

cannot be used, but unmoved fields can.

Destructuring and ownership: let Point { x, y } = point; moves x and y out of point. Pattern destructuring follows move semantics for non-Copy types.

Temporary lifetimes: temporaries created in expressions live until the end of the enclosing statement. Exception: temporaries in let bindings live for the block scope.

Reborrowing: &mut references can be implicitly reborrowed. A function taking &mut T can accept a &mut reference without explicit reborrowing. Reborrowing is transparent.

Two-phase borrowing: a mutable borrow can be split into: an initial borrow (for the receiver) and an activation (when mutation occurs). Two-phase borrowing enables: v.push(v.len()).

Non-lexical lifetimes (NLL): borrows end when they are last used, not at the end of the scope. NLL makes the borrow checker more precise and reduces false positives.

Drop flags: the compiler tracks whether a value has been moved at runtime using drop flags. Drop flags ensure correct destruction even with conditional moves.

Placement new: constructing values directly in their final location without intermediate moves. The compiler optimizes large struct construction to avoid copies.

Swap and replace: std::mem::swap exchanges two values. std::mem::replace replaces a value and returns the old one. These functions work with mutable references.

ManuallyDrop: ManuallyDrop[T] prevents automatic dropping. Used for: FFI, union fields, and custom drop logic. The value must be explicitly dropped with ManuallyDrop::drop.

8.1 Advanced Pattern Matching

Nested patterns: match value { Some(Ok(x)) => use(x), Some(Err(e)) => handle(e), None => default() }. Patterns can be nested to arbitrary depth.

Slice patterns: match slice { [first, .., last] => {}, [single] => {}, [] => {} }. Slice patterns match arrays and slices with: fixed elements, rest (..), and length.

Range patterns: match x { 0..=9 => 'digit', 'a'..='z' => 'lower', _ => 'other' }. Range patterns match inclusive ranges of integers or characters.

Const patterns: const ZERO: i32 = 0; match x { ZERO => {}, _ => {} }. Constants can be used as patterns. The constant value is compared for equality.

Binding modes: match &x { y => {} } automatically binds y as a reference. Without the automatic binding mode, the pattern would move x.

At-bindings: match x { y @ 1..=9 => use(y), _ => {} }. The @ operator binds the matched value to a variable while also testing a pattern.

Pattern compilation: the compiler compiles patterns to decision trees. Decision trees minimize the

number of comparisons. Unreachable arms are detected.

Irrefutable patterns: let (a, b) = pair; is irrefutable (always matches). Irrefutable patterns are used in: let, for, function parameters, and closure parameters.

Refutable patterns: if let Some(x) = opt { } is refutable (may not match). Refutable patterns are used in: if-let, while-let, and match.

Pattern exhaustiveness: the compiler checks exhaustiveness using a matrix-based algorithm. The algorithm handles: enums, integers, structs, tuples, and nested patterns.

9.1 Trait System Advanced

Blanket implementations: impl[T: Display] ToString for T { }. Blanket implementations provide trait implementations for all types satisfying bounds.

Negative implementations: impl !Send for RawPointer { }. Negative implementations explicitly opt out of auto-traits. Used for: raw pointers and interior mutability.

Auto traits: Send and Sync are auto traits. Auto traits are automatically implemented for types whose fields all implement the trait.

Marker traits: Copy, Sized, Unpin are marker traits. Marker traits have no methods. They indicate properties of types to the compiler.

Coherence rules: at most one implementation of a trait for a type. The orphan rule: either the trait or the type must be defined in the current crate.

Specialization (unstable): default fn method() allows overriding in more specific implementations. Specialization enables: optimized implementations for specific types.

Associated constants: trait HasDefault { const DEFAULT: Self; }. Associated constants provide type-level values. Implementors define the constant's value.

Generic associated types (GATs): trait Container { type Item[T]; }. GATs allow associated types with type parameters. Used for: lending iterators, streaming iterators.

Trait aliases: trait ReadWrite = Read + Write; creates an alias for a combination of traits. Trait aliases simplify complex trait bounds.

Trait resolution order: the compiler resolves trait methods by: inherent methods first, then trait methods in scope. Ambiguity between traits requires explicit disambiguation.

10.1 Generic Advanced Topics

Phantom types: struct Meters[T] { value: f64, _marker: PhantomData[T] }. Phantom types add type-level information without runtime cost. Used for: unit types, state machines.

Higher-kinded types (simulated): trait Functor { type HKT[T]; fn map[A, B](self: Self::HKT[A], f: fn(A)

-> B) -> Self::HKT[B]; }. HKTs are simulated using associated types.

Type-level computation: const generics enable type-level arithmetic: struct Matrix[const R: usize, const C: usize]. Type-level computation is evaluated at compile time.

Impl trait: fn make_iter() -> impl Iterator[Item = i32]. Impl trait in return position hides the concrete type. The compiler knows the concrete type; callers do not.

Existential types: type MyIter = impl Iterator[Item = i32]. Existential types name a hidden concrete type. Used for: return types, associated types, and async return types.

Turbofish syntax: function::[Type](args). Turbofish explicitly specifies type parameters when inference is insufficient: Vec::[i32]::new().

Where clauses: fn complex[T, U](t: T, u: U) where T: Clone + Debug, U: From[T] + Send. Where clauses allow complex bounds that do not fit inline.

Trait object generics: fn process(items: &[Box[dyn Display + Send]]) processes a collection of dynamically-typed items. Trait object generics combine dynamic dispatch with bounds.

Generic defaults: struct Container[T = String] { value: T }. Default type parameters reduce boilerplate. Container without type argument uses String.

Implied bounds: where T: Iterator implies that T::Item exists. The compiler derives additional bounds from trait definitions.

11.1 Concurrency Primitives

Atomic types: AtomicBool, AtomicI32, AtomicU64, AtomicPtr. Atomic operations: load, store, swap, compare_and_swap. Memory orderings: Relaxed, Acquire, Release, AcqRel, SeqCst.

Read-write locks: RwLock[T] allows multiple readers or one writer. read() returns a read guard. write() returns a write guard. RwLock prevents writer starvation.

Condvar: Condvar allows threads to wait for conditions. wait(guard) releases the lock and sleeps. notify_one() wakes one waiter. notify_all() wakes all waiters.

Barrier: Barrier synchronizes multiple threads at a point. wait() blocks until all threads have called wait(). Used for: phased computation, synchronized startup.

Once: Once ensures a function is called exactly once. call_once(|| { init() }) initializes once, even with concurrent callers. Used for: lazy static initialization.

Thread-local storage: thread_local! { static KEY: Cell[u32] = Cell::new(0); }. Each thread has an independent copy. Thread-local access is efficient (no synchronization).

Scoped threads: std::thread::scope(|s| { s.spawn(|| { borrow_stack_data() }); }). Scoped threads can borrow data from the spawning scope. The scope ensures all threads complete.

Parking: thread::park() suspends the current thread. thread::unpark() resumes a parked thread.

Parking is a low-level primitive for custom synchronization.

Yield: thread::yield_now() yields the CPU to other threads. Yielding is cooperative. Used in: spin loops, busy-waiting, and fair scheduling.

Thread naming: thread::Builder::new().name('worker'.to_string()).spawn(|| {}). Named threads appear in: debugger thread lists, panic messages, and profiler output.

12.1 Unsafe Detailed Rules

Unsafe contracts: unsafe functions document preconditions. Callers must satisfy preconditions. Violating preconditions is undefined behavior (UB).

Undefined behavior: UB includes: dereferencing null/dangling pointers, data races, invalid values (bool not 0/1), breaking aliasing rules, and unwinding into foreign code.

Unsafe traits: unsafe trait GlobalAlloc { }. Implementing unsafe traits has safety requirements. The implementor guarantees: correctness of the implementation.

Union types: union MyUnion { i: i32, f: f32 }. Accessing union fields requires unsafe. The caller must ensure the correct field is accessed.

Inline assembly: unsafe { asm!('mov {0}, {1}', out(reg) result, in(reg) input); }. Inline assembly allows direct machine code. The programmer ensures correctness.

Transmute: unsafe { std::mem::transmute::[i32, f32](bits) }. Transmute reinterprets bits as a different type. Transmute is extremely dangerous; prefer safe alternatives.

Raw pointer arithmetic: ptr.add(n), ptr.sub(n), ptr.offset(n). Pointer arithmetic must stay within the allocated object. Out-of-bounds arithmetic is UB.

Unsafe guidelines: minimize unsafe blocks. Document safety invariants. Encapsulate unsafe code in safe abstractions. Test thoroughly with: Miri, sanitizers, and fuzzing.

Miri: an interpreter that detects: UB, memory leaks, and data races in unsafe code. cargo miri test runs tests under Miri. Miri is essential for validating unsafe code.

Stacked borrows: Miri uses the stacked borrows model to detect aliasing violations. Stacked borrows track: tag-based permissions for each pointer. Violations indicate UB.

13.1 Standard Library Collections

Vec[T]: growable array. push(), pop(), insert(), remove(), iter(). Amortized O(1) push. Contiguous memory layout for cache efficiency.

HashMap[K, V]: hash map with Robin Hood hashing. insert(), get(), remove(), contains_key(). Average O(1) operations. Requires K: Hash + Eq.

BTreeMap[K, V]: B-tree sorted map. insert(), get(), remove(), range(). O(log n) operations. Sorted

iteration. Requires K: Ord.

HashSet[T]: hash set. insert(), contains(), remove(), intersection(), union(). Average O(1) operations. Requires T: Hash + Eq.

BTreeSet[T]: B-tree sorted set. insert(), contains(), remove(), range(). O(log n) operations. Sorted iteration. Requires T: Ord.

VecDeque[T]: double-ended queue using a ring buffer. push_front(), push_back(), pop_front(), pop_back(). O(1) operations at both ends.

LinkedList[T]: doubly-linked list. push_front(), push_back(), pop_front(), pop_back(). O(1) insertion/removal at endpoints. O(n) random access. Rarely preferred over Vec.

BinaryHeap[T]: max-heap. push(), pop(), peek(). O(log n) push/pop. O(1) peek. Used for: priority queues. Requires T: Ord.

Entry API: map.entry(key).or_insert(default). The entry API provides efficient insert-or-update. Variants: or_insert(), or_insert_with(), and_modify().

Iterators: all collections implement IntoIterator. iter() for shared references. iter_mut() for mutable references. into_iter() for owned values.

3.2 Reference Types Detail

Shared references: &T provides read-only access. Multiple shared references can coexist. Shared references are Copy. Shared references prevent mutation of the referent.

Mutable references: &mut T provides read-write access. At most one mutable reference to a value. Mutable references are not Copy. Mutable references have exclusive access.

Reference coercion: &mut T coerces to &T (downgrade). &T does not coerce to &mut T. Coercion allows passing &mut where & is expected.

Deref coercion: &String coerces to &str via Deref. &Vec[T] coerces to &[T]. &Box[T] coerces to &T. Deref coercion chains: &Box[String] -> &String -> &str.

Slice references: &[T] is a fat pointer (pointer + length). &str is a fat pointer to UTF-8 bytes. Fat pointers are twice the size of regular pointers.

Trait object references: &dyn Trait is a fat pointer (pointer + vtable). The vtable contains: method pointers, size, alignment, and drop function.

Reference lifetimes: every reference has a lifetime. The lifetime is the scope during which the reference is valid. The borrow checker enforces lifetime correctness.

Lifetime subtyping: 'a: 'b means lifetime 'a outlives 'b. A reference with lifetime 'a can be used where 'b is expected if 'a: 'b.

Static lifetime: 'static is the longest lifetime. String literals have type &'static str. 'static references are

valid for the entire program duration.

Anonymous lifetimes: fn process(s: &str) -> &str uses anonymous lifetimes. The compiler assigns lifetime parameters using elision rules.

4.2 Method Call Expressions

Method resolution: value.method() resolves by: checking inherent methods, then trait methods in scope. If ambiguous, a compile error suggests qualification.

Auto-ref and auto-deref: the compiler automatically adds &, &mut, or * to match the method receiver. value.len() may auto-ref to (&value).len().

Fully qualified syntax: <Type as Trait>::method(value). Fully qualified syntax disambiguates between traits. Required when multiple traits provide methods with the same name.

Associated function calls: Type::new(). Associated functions do not have a self receiver. They are called on the type, not on a value.

Method chaining: value.method1().method2().method3(). Methods that return Self or a related type enable chaining. Chaining produces fluent APIs.

Extension methods via traits: trait VecExt { fn sorted(self) -> Self; } impl VecExt for Vec[i32] { fn sorted(mut self) -> Self { self.sort(); self } }. Extension methods add behavior to existing types.

Method visibility: methods follow the same visibility rules as other items. pub methods are accessible from outside the module. Private methods are implementation details.

Generic methods: impl Container { fn get[T: Into[Key]](&self, key: T) -> Option[&Value] }. Generic methods accept multiple input types via trait bounds.

Async methods: impl Server { async fn handle(&self, req: Request) -> Response }. Async methods return futures. Called with: server.handle(req).await.

Const methods: impl Type { const fn size() -> usize { mem::size_of::[Self]() } }. Const methods can be called at compile time. Used for: const evaluation.

6.2 Iterator Expressions

Iterator trait: trait Iterator { type Item; fn next(&mut self) -> Option[Self::Item]; }. Iterators produce a sequence of values. next() returns None when exhausted.

Iterator adaptors: map(f), filter(f), flat_map(f), take(n), skip(n), zip(other), chain(other), enumerate(), peekable(), cloned(), copied().

Iterator consumers: collect(), for_each(f), fold(init, f), sum(), product(), count(), min(), max(), any(f), all(f), find(f), position(f).

Lazy evaluation: adaptors are lazy; they produce values on demand. No computation occurs until a

consumer is called. Lazy evaluation avoids unnecessary work.

Double-ended iterators: DoubleEndedIterator provides next_back(). Enables: rev() (reverse iteration), rfold(), rfind(). VecDeque and slice iterators are double-ended.

Exact-size iterators: ExactSizeIterator provides len(). Used for: pre-allocation, progress reporting. Vec, array, and range iterators have exact size.

Fused iterators: FusedIterator guarantees that after returning None, all subsequent calls return None. Most standard iterators are fused.

Custom iterators: struct Counter { count: u32 } impl Iterator for Counter { type Item = u32; fn next(&mut self) -> Option[u32] { self.count += 1; Some(self.count) } }.

Into-iterator: IntoIterator converts a collection into an iterator. for x in collection desugars to: for x in IntoIterator::into_iter(collection).

Parallel iterators: the rayon crate provides parallel iterators. par_iter() creates a parallel iterator. Parallel iteration uses a work-stealing thread pool.

7.2 Borrow Checker Details

Borrow scope: a borrow starts at the borrow expression and ends at the last use of the reference. Non-lexical lifetimes (NLL) enable precise borrow scoping.

Borrow conflict: &mut x conflicts with any other borrow of x. &x does not conflict with other &x borrows. Conflicts are compile errors.

Interior mutability: Cell[T] and RefCell[T] allow mutation through shared references. Cell uses copy semantics. RefCell uses runtime borrow checking. UnsafeCell is the primitive.

Pin: Pin[&mut T] prevents moving the value. Pinning is required for: self-referential structs, futures, and async state machines.

Borrow splitting: the borrow checker can split borrows to different struct fields. &mut point.x and &mut point.y can coexist because they borrow different fields.

Borrow and closures: closures that capture &mut references are FnMut. Closures that capture &references are Fn. The borrow checker enforces closure capture rules.

Borrow and matches: match value { ref x => {} } borrows value instead of moving. ref mut x borrows mutably. Match borrows follow the same rules as explicit borrows.

Borrow diagnostics: the compiler provides detailed borrow error messages: conflicting borrows, moved values, and lifetime insufficiency. Suggestions include code fixes.

Polonius: the next-generation borrow checker (Polonius) uses a more precise analysis based on dataflow. Polonius accepts more valid programs than the current NLL checker.

Borrow checker escape: unsafe code can bypass the borrow checker. Raw pointers have no borrow

checking. Unsafe code must manually ensure aliasing and lifetime correctness.

9.2 Standard Traits

Display: formats a value for user-facing output. impl Display for Type { fn fmt(&self, f: &mut Formatter) -> fmt::Result { write!(f, ...) } }. Used by: println! and format!.

Debug: formats a value for debugging. #[derive(Debug)] auto-generates Debug. Debug output includes: type name, field names, and field values. Used by: dbg! and {:?}.

Clone: creates a deep copy. #[derive(Clone)] auto-generates Clone. clone() returns a new independent copy. Clone may allocate memory.

Copy: marker trait for bitwise copy. #[derive(Copy, Clone)] auto-generates both. Copy types are duplicated on assignment. Only types with no heap allocation can be Copy.

PartialEq and Eq: equality comparison. #[derive(PartialEq, Eq)]. PartialEq allows NaN != NaN. Eq guarantees reflexivity (x == x for all x).

PartialOrd and Ord: ordering comparison. #[derive(PartialOrd, Ord)]. PartialOrd allows incomparable values. Ord guarantees total ordering.

Hash: produces a hash value. #[derive(Hash)]. Hash is used by: HashMap, HashSet. Types implementing Eq and Hash must satisfy: if a == b, then hash(a) == hash(b).

Default: provides a default value. #[derive(Default)]. Default::default() creates the default. Numeric defaults: 0. Bool default: false. Option default: None.

From and Into: type conversion. impl From[String] for MyType { }. Into is automatically implemented when From is implemented. The ? operator uses From for error conversion.

Drop: custom destructor. impl Drop for Type { fn drop(&mut self) { cleanup() } }. Drop is called automatically when a value goes out of scope.

11.2 Async Runtime Details

Future trait: trait Future { type Output; fn poll(self: Pin[&mut Self], cx: &mut Context) -> Poll[Self::Output]; }. Futures are state machines that can be polled for completion.

Poll enum: Poll::Ready(value) indicates completion. Poll::Pending indicates the future is not ready. The runtime re-polls when notified via the Waker.

Waker: the Waker notifies the runtime to re-poll a future. Wakers are registered with I/O sources. When data is available, the waker is called, and the runtime re-polls.

Executor: the executor drives futures to completion. Single-threaded: run futures on one thread. Multi-threaded: distribute futures across a thread pool.

Reactor: the reactor monitors I/O sources (sockets, files, timers). When an event occurs, the reactor

wakes the associated future. Reactors use: epoll (Linux), kqueue (macOS).

Task: a task is a top-level future managed by the executor. spawn(future) creates a task. Tasks run concurrently. JoinHandle retrieves the task's output.

Select: select! { result = future1 => {}, result = future2 => {} }. Select waits for the first future to complete. Unfinished futures are cancelled.

Join: join!(future1, future2) runs futures concurrently and waits for all to complete. Returns a tuple of results.

Stream: trait Stream { type Item; fn poll_next(self: Pin[&mut Self], cx: &mut Context) -> Poll[Option[Self::Item]]; }. Streams are async iterators.

Async cancellation: dropping a future cancels it. Cancellation is cooperative: the future must reach an .await point to be cancelled. Cleanup runs via Drop.

12.2 FFI Details

C calling convention: extern 'C' fn(x: i32) -> i32. The C ABI defines: argument passing (registers and stack), return value passing, and callee-saved registers.

Opaque types: extern { type OpaqueHandle; }. Opaque types represent C types whose layout is unknown. Opaque types are accessed only through pointers.

Callback functions: type Callback = extern 'C' fn(i32, *mut c_void); extern 'C' { fn register_callback(cb: Callback, data: *mut c_void); }. Callbacks pass Lateralus functions to C.

String FFI: CStr (borrowed C string), CString (owned C string). CString::new('hello') creates a null-terminated string. CStr::from_ptr(c_str) wraps a C string pointer.

Error FFI: C functions return error codes. Lateralus wrappers convert error codes to Result. Common pattern: if ret < 0 { Err(io::Error::last_os_error()) } else { Ok(ret) }.

Struct FFI: #[repr(C)] struct CStruct { x: i32, y: f64 }. repr(C) ensures C-compatible layout. repr(packed) removes padding. repr(align(N)) specifies alignment.

Enum FFI: #[repr(i32)] enum CEnum { A = 0, B = 1, C = 2 }. repr(i32) ensures the enum uses a 32-bit integer representation. Compatible with C enums.

Global FFI: extern 'C' { static GLOBAL: i32; }. Foreign globals are accessed with unsafe. The linker resolves the symbol. Global access is safe if the foreign library guarantees validity.

Variadic FFI: extern 'C' { fn printf(fmt: *const c_char, ...) -> i32; }. Variadic functions accept a variable number of arguments. Calling variadic functions requires unsafe.

Panic and FFI: panicking across FFI boundaries is undefined behavior. Use catch_unwind at FFI boundaries. Return error codes instead of panicking.

13.2 Standard Library I/O

Read trait: fn read(&mut self, buf: &mut [u8]) -> io::Result[usize]. Read provides byte-level input. Implementations: File, TcpStream, stdin, Cursor, BufReader.

Write trait: fn write(&mut self, buf: &[u8]) -> io::Result[usize]. Write provides byte-level output. Implementations: File, TcpStream, stdout, stderr, BufWriter, Vec[u8].

BufRead trait: fn read_line(&mut self, buf: &mut String) -> io::Result[usize]. BufRead adds buffered reading. lines() returns an iterator over lines.

Seek trait: fn seek(&mut self, pos: SeekFrom) -> io::Result[u64]. Seek repositions the read/write cursor. SeekFrom: Start(n), End(n), Current(n).

File operations: File::open(path), File::create(path), OpenOptions::new().read(true).write(true).open(path). File implements: Read, Write, Seek.

Buffered I/O: BufReader::new(reader) wraps a reader with a buffer. BufWriter::new(writer) wraps a writer. Buffering reduces system call overhead.

Formatted output: write!(f, 'value: {}', v) writes formatted output. writeln! adds a newline. Formatting uses the Display trait by default, Debug with {:?}.

Standard streams: io::stdin(), io::stdout(), io::stderr(). Standard streams are thread-safe. Locking: stdin().lock() returns a locked handle for efficient repeated reads.

Pipe composition: let output = Command::new('ls').stdout(Stdio::piped()).spawn()?.wait_with_output()?. Process I/O can be piped to/from Lateralus streams.

Async I/O: tokio::fs::File, tokio::net::TcpStream. Async I/O avoids blocking threads. AsyncRead and AsyncWrite mirror the synchronous traits.

2.2 Macro Syntax

Declarative macros: macro_rules! vec { ($($elem:expr),*) => { { let mut v = Vec::new(); $(v.push($elem);)* v } }; }. Pattern matching on token trees.

Fragment specifiers: $name:expr (expression), $name:ty (type), $name:ident (identifier), $name:pat (pattern), $name:stmt (statement), $name:tt (token tree).

Repetition: $($name:expr),* matches zero or more comma-separated expressions. $($name:expr),+ matches one or more. $($name:expr);* uses semicolons.

Macro hygiene: macros introduce names that do not collide with the expansion context. Hygienic macros prevent accidental variable capture. Lateralus macros are partially hygienic.

Procedural macros: proc-macro crates define macros as functions. fn my_macro(input: TokenStream) -> TokenStream. Proc-macros enable: derive macros, attribute macros, function-like macros.

Derive macros: #[derive(MyTrait)] invokes a proc-macro. The macro receives the struct/enum definition. The macro generates an impl block.

Attribute macros: #[my_attr] fn item() { }. The macro receives the attribute arguments and the annotated item. The macro returns the transformed item.

Token streams: TokenStream is a sequence of token trees. Token trees: identifiers, literals, punctuation, and groups (delimited by () [] {}). Token streams are the proc-macro API.

Macro debugging: cargo expand shows fully expanded source. #[trace_macros] logs macro expansion steps. Macro debugging is essential for complex macros.

Macro best practices: prefer functions over macros when possible. Use macros for: compile-time code generation, DSLs, and repetitive patterns that cannot be abstracted with generics.

3.3 Algebraic Data Types

Product types (structs): struct Point { x: f64, y: f64 }. Structs combine multiple values. Named fields: Point { x: 1.0, y: 2.0 }. Tuple structs: struct Color(u8, u8, u8).

Sum types (enums): enum Shape { Circle(f64), Rectangle(f64, f64), Triangle(f64, f64, f64) }. Enums represent one of several variants. Each variant can hold data.

Unit structs: struct Marker;. Unit structs have no fields. Used for: type-level markers, trait implementations, and zero-sized types.

Recursive types: enum List[T] { Cons(T, Box[List[T]]), Nil }. Recursive types require indirection (Box) because the compiler needs to know the type's size.

Generic ADTs: enum Option[T] { Some(T), None }. enum Result[T, E] { Ok(T), Err(E) }. Generic ADTs are the foundation of Lateralus error handling.

Newtype pattern: struct Miles(f64);. Newtypes wrap existing types. The newtype has: different identity, different trait implementations, and zero runtime overhead.

Enum discriminant: #[repr(u8)] enum Color { Red = 0, Green = 1, Blue = 2 }. Discriminants specify the enum's representation. Used for: FFI and serialization.

Struct update syntax: let p2 = Point { x: 5.0, ..p1 };. The update syntax copies remaining fields from another struct. Used for: defaults and partial updates.

Enum methods: impl Shape { fn area(&self) -> f64 { match self { Circle(r) => PI * r * r, Rectangle(w, h) => w * h, ... } } }. Enum methods use pattern matching.

Visibility: struct fields and enum variants can have individual visibility modifiers. pub struct Config { pub name: String, secret: String }. Private fields require a constructor.

5.2 Expression Evaluation

Evaluation order: function arguments are evaluated left to right. Operator operands are evaluated left to right. Short-circuit operators may skip the right operand.

Operator precedence (high to low): paths, method calls, field access, function calls, unary (! - & &mut *), as, * / %%, + -, << >>, &, ^, |, == != < > <= >=, &&, ||, .., =.

Temporary values: temporaries are created for intermediate expression results. Temporaries live until the end of the enclosing statement (or longer in let bindings).

Place expressions: expressions that refer to memory locations. Variables, field access, index, deref are place expressions. Place expressions can appear on the left side of assignment.

Value expressions: expressions that produce values. Literals, function calls, and operators are value expressions. Value expressions can only appear on the right side of assignment.

Coercion: implicit type conversion in specific contexts. Coercion sites: let statements, function arguments, struct fields, and return values. Examples: &mut to &, Box to &.

Overflow behavior: integer overflow panics in debug mode. In release mode, overflow wraps. The wrapping behavior is two's complement for signed integers.

Evaluation of const expressions: const expressions are evaluated at compile time. Const contexts include: const/static initializers, array lengths, and const fn calls.

Diverging expressions: expressions of type ! are compatible with any type. This allows: let x: i32 = if cond { 42 } else { panic!() };. The panic arm has type !.

Expression typing: every expression has a unique type determined at compile time. Type inference propagates type information bidirectionally (forward and backward).

8.2 Pattern Matching Optimization

Decision tree compilation: the compiler converts patterns into decision trees. Decision trees minimize comparisons by: testing discriminants first, then fields.

Branch reordering: the compiler reorders branches for: cache efficiency, branch prediction, and minimum comparisons. Frequently-taken branches are placed first.

Constant folding in patterns: const values in patterns are evaluated at compile time. No runtime comparison is needed for unreachable constant patterns.

Range splitting: range patterns (0..=9) are compiled to: comparison against lower bound, comparison against upper bound. Combined into a single subtraction and comparison.

Switch tables: dense integer patterns are compiled to jump tables. Jump tables provide O(1) dispatch. Sparse patterns use binary search or linear search.

String pattern optimization: string patterns are compiled to: length check, then byte comparison. Short strings use inline comparison. Long strings use memcmp.

Nested pattern flattening: nested patterns are flattened into a single decision tree. Flattening avoids: redundant comparisons, redundant loads, and redundant branches.

Usefulness checking: the compiler checks whether each pattern is useful (can match a value not matched by previous patterns). Useless patterns produce warnings.

Exhaustiveness checking: the compiler verifies all possible values are covered. Missing patterns produce errors. The algorithm handles: enums, integers, structs, and wildcards.

Pattern performance: pattern matching compiles to code equivalent to hand-written if-else chains. The compiler's optimization ensures: zero overhead compared to manual dispatch.

10.2 Generic Compilation

Monomorphization: each generic function is compiled once per unique set of type arguments. foo[i32]() and foo[String]() generate separate machine code.

Code bloat: monomorphization can increase binary size. Mitigation: the compiler merges identical monomorphizations. Shared code for types with the same layout.

Compile time: generics increase compile time proportionally to the number of monomorphizations. Heavy generic use (many types) slows compilation.

Dynamic dispatch alternative: dyn Trait uses a vtable instead of monomorphization. One copy of the code handles all types. Trade-off: runtime overhead vs binary size.

Generic bounds checking: the compiler verifies trait bounds at the call site. Missing bounds produce compile errors with: expected traits, available implementations, and suggestions.

Inference with generics: the compiler infers type parameters from: function arguments, return type context, and method calls. Turbofish syntax disambiguates when inference fails.

Const generic evaluation: const generic parameters are evaluated at compile time. Const evaluation supports: arithmetic, boolean logic, and array indexing. Complex expressions may not be supported.

Generic layout: generic types have layout determined by their type arguments. Vec[i32] and Vec[f64] have different element sizes. The compiler computes layout for each monomorphization.

Generic trait implementations: impl[T: Display] Printable for Vec[T] provides a generic trait implementation. The implementation is monomorphized for each T.

Zero-cost abstraction: generic abstractions compile to code identical to hand-written specialized code. Iterator chains, closures, and generic containers have no runtime overhead.

13.3 Standard Library Concurrency

Thread module: thread::spawn(closure) creates a new thread. JoinHandle::join() waits for completion. thread::sleep(duration) pauses the current thread.

Mutex[T]: mutual exclusion lock. lock() blocks until the mutex is available. try_lock() returns Err if the mutex is locked. Poisoned mutex indicates a panicked holder.

Arc[T]: atomic reference-counted pointer. Arc enables shared ownership across threads. Arc::clone() increments the reference count. Drop decrements and frees when zero.

Channel module: mpsc::channel() creates an unbounded channel. mpsc::sync_channel(n) creates a bounded channel. Bounded channels provide backpressure.

Atomic module: AtomicBool, AtomicUsize, AtomicPtr provide lock-free operations. Ordering: Relaxed (weakest), Acquire/Release (paired), SeqCst (strongest, sequential consistency).

Rayon crate: parallel iteration with: par_iter(), par_bridge(), par_sort(). Rayon uses work-stealing for load balancing. Rayon parallelism is automatic and safe.

Crossbeam crate: advanced concurrency primitives. crossbeam::channel (multi-producer, multi-consumer), crossbeam::epoch (epoch-based reclamation), crossbeam::deque (work-stealing).

Tokio runtime: async runtime for I/O-bound workloads. tokio::spawn() creates an async task. tokio::select! waits for the first ready future. Tokio provides: timers, I/O, and synchronization.

Async synchronization: tokio::sync::Mutex (async mutex), tokio::sync::RwLock (async read-write lock), tokio::sync::Semaphore (async semaphore), tokio::sync::Notify (async condition).

Concurrent collections: dashmap (concurrent hash map), flurry (concurrent hash map based on Java's ConcurrentHashMap). Lock-free data structures avoid mutex overhead.

14.1 Language Evolution

Edition system: Lateralus editions (2021, 2024, 2027) introduce backward-incompatible changes. Each crate specifies its edition. Different editions interoperate within a project.

Edition migration: lateralus-fix --edition 2024 automatically migrates code. The tool handles: syntax changes, import changes, and deprecated items. Migration is low-risk.

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

RFC process: language changes follow the RFC process. RFCs are: proposed, discussed, revised, and accepted. Accepted RFCs are implemented and stabilized.

Stability promise: stable Lateralus guarantees backward compatibility. Code that compiles on version N compiles on version N+1. Exceptions: soundness fixes and compiler bugs.

Deprecation: deprecated features emit warnings. Deprecated features are removed in the next edition. The deprecation period is at least one edition cycle (3 years).

Nightly features: experimental features are available on nightly. Nightly features may change or be removed. Nightly is for: experimentation, feedback, and early adoption.

Compiler versioning: the compiler follows semantic versioning. Patch releases: bug fixes. Minor releases: new features. Major releases: new editions.

Language team: the language team steers language design. Team responsibilities: RFC review, design decisions, stability, and community engagement.

Community involvement: language development is open to community participation. Contribution paths: RFCs, issue triage, implementation, documentation, and testing.

2.3 Whitespace and Tokens

Whitespace: spaces, tabs, newlines, and carriage returns are whitespace. Whitespace separates tokens but is otherwise insignificant. The parser ignores whitespace between tokens.

Token categories: keywords, identifiers, literals, operators, delimiters, and comments. The lexer classifies each token into a category for the parser.

Delimiter tokens: ( ) parentheses, [ ] brackets, { } braces. Delimiters must be balanced. Unbalanced delimiters produce parse errors.

Punctuation tokens: , (comma), ; (semicolon), : (colon), :: (path separator), -> (arrow), => (fat arrow), . (dot), .. (range), ..= (inclusive range), ... (rest pattern).

Lifetime tokens: 'a, 'b, 'static. Lifetime tokens start with a single quote followed by an identifier. Lifetimes annotate reference types and trait bounds.

Attribute tokens: #[...] outer attribute, #![...] inner attribute. Attributes provide metadata for items, expressions, and modules.

Doc comment tokens: /// generates #[doc = '...'] for the next item. //! generates #![doc = '...'] for the enclosing item. Doc comments support Markdown.

Raw identifiers: r#keyword allows using keywords as identifiers. Useful for: FFI (foreign function names), and compatibility with older code.

Unicode identifiers: identifiers can contain Unicode letters. Non-ASCII identifiers follow UAX #31. Confusable characters (Cyrillic a vs Latin a) produce warnings.

Shebang: #! at the start of a file is a shebang line (Unix executable scripts). The lexer ignores the shebang line.

4.3 Struct and Enum Expressions

Struct creation: let p = Point { x: 1.0, y: 2.0 };. Named fields are specified by name. Field init shorthand: let x = 1.0; let p = Point { x, y: 2.0 };

Tuple struct creation: let c = Color(255, 0, 0);. Tuple structs use positional arguments. Tuple structs with one field are newtype wrappers.

Unit struct creation: let m = Marker;. Unit structs have no fields. Unit struct creation uses the type name directly.

Functional update: let p2 = Point { x: 5.0, ..p1 };. Remaining fields are copied from p1. The source must have the same type. Non-Copy fields are moved.

Enum creation: let s = Shape::Circle(5.0);. Enum variants are created using the qualified variant name. Variants with data pass arguments.

Tuple enum variants: let color = Color::RGB(255, 0, 0);. Tuple variants use positional arguments like tuple structs.

Struct enum variants: let event = Event::Click { x: 10, y: 20 };. Struct variants use named fields like regular structs.

Unit enum variants: let dir = Direction::North;. Unit variants have no data. Unit variants are the simplest enum form.

Const struct creation: const ORIGIN: Point = Point { x: 0.0, y: 0.0 };. Struct creation in const context requires all fields to be const-evaluable.

Enum variant coercion: enum variants are not types; they are constructors. Shape::Circle is a function fn(f64) -> Shape. This enables: .map(Shape::Circle).

6.3 Error Handling Expressions

Result type: enum Result[T, E] { Ok(T), Err(E) }. Functions that can fail return Result. Ok wraps the success value. Err wraps the error.

The ? operator: let value = risky_function()?;. If the function returns Err, ? returns early with the error. If Ok, ? unwraps the value.

Error conversion: ? uses From::from to convert error types. impl From[io::Error] for MyError enables: io_function()? in functions returning Result[T, MyError].

Panic: panic!('message') terminates the current thread. Panics unwind the stack (calling Drop) or abort (configurable). Panics indicate unrecoverable errors.

Unwrap: result.unwrap() returns the Ok value or panics on Err. option.unwrap() returns Some value or panics on None. Unwrap is for prototyping; production code uses ?.

Expect: result.expect('context') like unwrap but with a custom panic message. Expect provides better error messages than unwrap.

Custom error types: enum AppError { NotFound, Unauthorized, Internal(String) }. impl Display for AppError. impl Error for AppError. impl From[io::Error] for AppError.

Error chaining: error.source() returns the underlying cause. anyhow::Error wraps errors with context: operation().context('failed to process')?;

Try blocks (unstable): let result: Result[i32, Error] = try { let x = parse()?; x + 1 };. Try blocks allow ? inside expressions.

Error handling patterns: map_err(|e| convert(e)) converts error types. unwrap_or(default) provides a default value. unwrap_or_else(|| compute()) provides a lazy default.

7.3 Lifetime Advanced Topics

Lifetime bounds: T: 'a means T outlives lifetime 'a. Required for: references inside structs, trait objects, and closures. Ensures the value is valid for the lifetime.

Higher-ranked trait bounds (HRTB): for['a] Fn(&'a str) -> &'a str. HRTB quantifies over all possible lifetimes. Required for: closures accepting references.

Lifetime variance: &'a T is covariant in 'a (can shorten). &'a mut T is invariant in 'a (cannot change). fn(&'a T) is contravariant in 'a (can lengthen).

Lifetime in struct: struct Ref['a, T] { value: &'a T }. The struct cannot outlive the referenced data. Multiple lifetime parameters allow different reference lifetimes.

Lifetime elision in impl: impl['a] Ref['a, str] { fn get(&self) -> &str { self.value } }. The compiler elides the output lifetime as &'a str because of the &self receiver.

Named lifetime in trait: trait Parser['a] { fn parse(&self, input: &'a str) -> &'a str; }. Lifetime parameters on traits constrain implementors.

Outlives relation: 'a: 'b means lifetime 'a is at least as long as 'b. Used in: where clauses, struct definitions, and trait bounds.

Lifetime and trait objects: Box[dyn Trait + 'a] is a trait object valid for lifetime 'a. Box[dyn Trait + 'static] is valid for the entire program.

Lifetime and closures: closures capture references with the captured variable's lifetime. move closures take ownership, removing lifetime constraints.

Lifetime debugging: the compiler provides detailed lifetime error messages. Messages include: expected lifetime, actual lifetime, and the conflicting references.

9.3 Trait Implementation Patterns

Delegation pattern: impl Display for Wrapper { fn fmt(&self, f: &mut Formatter) -> fmt::Result { self.inner.fmt(f) } }. Delegate trait methods to inner fields.

Newtype trait implementation: struct MyVec(Vec[i32]); impl Deref for MyVec { type Target = Vec[i32]; fn deref(&self) -> &Vec[i32] { &self.0 } }. Newtype wraps to add traits.

Conditional implementation: impl[T: Display] Printable for Option[T] { }. Implement traits conditionally based on type parameter bounds.

Blanket implementation: impl[T: Debug + Display] Log for T { fn log(&self) { println!('{}', self); } }. Blanket implementations cover all qualifying types.

Orphan rule workaround: define a newtype wrapper to implement foreign traits on foreign types. struct MyWrapper(ForeignType); impl ForeignTrait for MyWrapper { }.

Default methods: trait Logger { fn log(&self, msg: &str); fn warn(&self, msg: &str) { self.log(&format!('WARN: {}', msg)); } }. Default methods provide reusable behavior.

Sealed traits: mod private { pub trait Sealed {} } pub trait Public: private::Sealed { }. External code cannot implement Public because Sealed is private.

Extension traits: trait IteratorExt: Iterator { fn sorted(self) -> Vec[Self::Item] where Self::Item: Ord; }. Extension traits add methods to existing traits.

Marker trait implementation: impl Copy for Point {}. impl Eq for Point {}. Marker traits have no methods; they indicate type properties.

Trait object construction: let printable: Box[dyn Display] = Box::new(42); let reader: &dyn Read = &file;. Trait objects are constructed by coercion from concrete types.

11.3 Memory Ordering

Relaxed ordering: no synchronization guarantees. Only guarantees: atomicity. Used for: counters, statistics. Not suitable for: communication between threads.

Acquire ordering: a load with Acquire sees all writes from the corresponding Release store. Used for: reading shared state after a flag is set.

Release ordering: a store with Release is visible to subsequent Acquire loads. Used for: publishing shared state, setting flags.

AcqRel ordering: combines Acquire and Release. A read-modify-write operation with AcqRel acquires on read and releases on write.

SeqCst ordering: sequential consistency. All SeqCst operations form a single total order agreed upon by all threads. Strongest ordering. Highest overhead.

Ordering trade-offs: weaker orderings allow more hardware optimization. Stronger orderings provide more guarantees. Choose the weakest ordering that is correct.

Fence: atomic::fence(Ordering) establishes ordering without an atomic operation. Fences are used for: batch synchronization, custom lock implementations.

Compare-and-swap: atomic.compare_exchange(current, new, success_order, failure_order). CAS is the fundamental building block for lock-free algorithms.

Fetch operations: fetch_add, fetch_sub, fetch_and, fetch_or, fetch_xor. Fetch operations atomically modify and return the previous value.

Hardware mapping: Relaxed maps to plain load/store (x86). Acquire maps to load with acquire semantics (ARM: LDAR). Release maps to store with release semantics (ARM: STLR).

12.3 Unsafe Abstractions

Safe wrapper pattern: unsafe { raw_op() } is wrapped in a safe function. The safe function documents: preconditions validated by the wrapper, invariants maintained, and UB prevented.

Unsafe trait: unsafe trait GlobalAlloc { unsafe fn alloc(&self, layout: Layout) -> *mut u8; }. Both declaring and implementing unsafe traits requires: safety documentation.

Interior mutability: UnsafeCell[T] is the primitive for interior mutability. Cell[T] wraps UnsafeCell with copy semantics. RefCell[T] wraps UnsafeCell with runtime borrow checking.

Unsafe field access: union fields require unsafe to access. Raw pointer fields do not require unsafe to read the pointer, but do require unsafe to dereference.

Custom allocator: unsafe impl GlobalAlloc for MyAlloc { unsafe fn alloc(&self, layout: Layout) -> *mut u8 { sys::alloc(layout.size()) } }. Custom allocators manage memory.

Unsafe and threads: data races are UB. Unsafe code that modifies shared state must use: atomics, locks, or other synchronization. Send and Sync enforce this at compile time.

Unsafe and drop: manually calling drop on a value and then using it is UB. ManuallyDrop prevents automatic drop. Unsafe code must track drop state.

Unsafe and alignment: accessing unaligned pointers is UB on some architectures. read_unaligned and write_unaligned handle unaligned access. Alignment is specified by #[repr(align(N))].

Unsafe audit: unsafe blocks should be: minimal (contain only the unsafe operation), documented (explain why it is safe), and tested (with Miri, sanitizers, and fuzzing).

Unsafe in standard library: the standard library uses unsafe extensively for: performance, FFI, and low-level operations. All unsafe code in std is: reviewed, documented, and tested.

13.4 Standard Library Path and Filesystem

Path: std::path::Path is an unsized type representing a file path. &Path is a reference to a path. Path is: platform-aware (/ on Unix, \ on Windows).

PathBuf: owned, mutable path. PathBuf : Path as String : str. PathBuf::from('/usr/bin') creates a path. push() appends components.

Path operations: parent() (parent directory), file_name() (last component), extension() (file extension), stem() (file name without extension), join() (append path).

Canonicalization: Path::canonicalize() resolves: symlinks, . (current), .. (parent), and returns the absolute path. Canonicalization may fail if the path does not exist.

Filesystem operations: fs::read_to_string(path), fs::write(path, data), fs::create_dir_all(path), fs::remove_file(path), fs::remove_dir_all(path).

Directory traversal: fs::read_dir(path) returns an iterator of DirEntry. Each entry has: path(), file_name(), file_type(), and metadata().

Metadata: fs::metadata(path) returns: file size, permissions, modification time, creation time, and file type (file, directory, symlink).

Permissions: metadata.permissions() returns permissions. set_readonly(true) makes a file read-only. Unix-specific: mode() returns the Unix permission bits.

Temporary files: tempfile crate provides: NamedTempFile (named temporary file), TempDir (temporary directory). Temp files are deleted when dropped.

Watching: notify crate provides filesystem watching. RecommendedWatcher monitors: create, modify, delete, rename events. Watching enables: live reload, build systems.

4.4 Unsafe Expressions

Unsafe block: unsafe { expr }. Unsafe blocks enable: raw pointer dereference, mutable static access, unsafe function calls, and union field access.

Raw pointer creation: &x as *const T (from reference), ptr::null() (null pointer), ptr::null_mut() (null mutable). Creating raw pointers is safe; dereferencing is unsafe.

Raw pointer dereference: unsafe { *ptr }. The pointer must be: non-null, aligned, pointing to valid memory, and respecting aliasing rules.

Unsafe function call: unsafe { libc::malloc(size) }. Unsafe functions have preconditions that the caller must satisfy. Preconditions are documented.

Mutable static access: unsafe { COUNTER += 1; }. Mutable statics can cause data races. Access must be synchronized (atomics or locks).

Union field access: unsafe { my_union.field }. The caller must ensure the union currently holds the accessed field. Incorrect access is UB.

Inline assembly: unsafe { asm!('nop'); }. Assembly instructions are platform-specific. The compiler cannot verify assembly correctness.

Transmute: unsafe { mem::transmute::[u32, f32](bits) }. Transmute reinterprets memory. The source and target types must have the same size.

offset and offset_from: unsafe { ptr.offset(n) }. Pointer arithmetic must stay within the allocation. offset_from computes the distance between pointers.

Volatile access: unsafe { ptr::read_volatile(addr) }. Volatile prevents compiler optimization of memory access. Used for: memory-mapped I/O, hardware registers.

6.4 Pipeline Expressions Detail

Pipeline syntax: expr |> fn is equivalent to fn(expr). Multiple pipelines: expr |> f |> g is equivalent to g(f(expr)). Pipelines are left-associative.

Pipeline with methods: expr |> .method() calls expr.method(). The dot prefix indicates a method call. Pipelines mix functions and methods.

Pipeline with closures: expr |> |x| x + 1 applies a closure. Closures in pipelines enable: inline transformation, filtering, and mapping.

Pipeline error handling: expr |> fn? propagates errors through the pipeline. The ? operator works at each pipeline stage.

Pipeline type inference: the compiler infers types at each pipeline stage. Explicit types can be annotated: expr |> fn::[Type].

Pipeline and iterators: (1..10) |> .filter(|x| x %% 2 == 0) |> .map(|x| x * x) |> .collect::[Vec[i32]](). Pipelines naturally compose iterator operations.

Pipeline performance: pipelines compile to the same code as nested function calls. No runtime overhead. The pipeline operator is purely syntactic.

Pipeline vs method chaining: pipelines work with free functions. Method chaining requires methods on the type. Pipelines are more flexible for: function composition.

Pipeline debugging: each pipeline stage can be inspected with: |> dbg!. dbg! prints the value and passes it through, enabling: non-intrusive debugging.

Pipeline composition: let process = compose!(parse, validate, transform); data |> process. Pipeline composition creates reusable transformations.

14.2 Appendix: Grammar Summary

Program: item*. A program is a sequence of items. Items include: functions, structs, enums, traits, implementations, modules, use declarations, and constants.

Function: visibility? 'fn' IDENT generics? '(' params? ')' ('->' type)? where-clause? block. Functions are the primary computation unit.

Struct: visibility? 'struct' IDENT generics? ('{' fields '}' | '(' types ')' ';' | ';'). Three forms: named, tuple, unit.

Enum: visibility? 'enum' IDENT generics? '{' variants '}'. Variants: IDENT ('(' types ')' | '{' fields '}' | empty).

Trait: visibility? 'trait' IDENT generics? (':' bounds)? '{' trait-items '}'. Trait items: methods, associated types, associated constants.

Impl: 'impl' generics? type '{' impl-items '}' | 'impl' generics? trait 'for' type '{' impl-items '}'. Two forms:

inherent and trait.

Expression: literal | path | block | if | match | loop | for | while | closure | call | field | index | unary | binary | pipeline | return | break | continue.

Statement: let-statement | expression-statement | item-statement. Statements end with semicolons (except block expressions).

Pattern: literal | ident | wildcard | tuple | struct | enum | reference | range | or | guard | at-binding. Patterns appear in: let, match, if-let, for.

Type: path-type | reference-type | array-type | slice-type | tuple-type | function-type | trait-object-type | impl-trait-type | never-type.

A.1 Appendix: Operator Precedence Table

Precedence level 1 (highest): field access (.), method call (.method()), function call (f()), indexing ([]).

Precedence level 2: unary prefix operators: - (negation), ! (logical not), & (borrow), &mut (mutable borrow), * (dereference).

Precedence level 3: type cast: as. Casting converts between numeric types. Casting to bool is not allowed. Casting pointers changes the pointed-to type.

Precedence level 4: multiplicative: * (multiply), / (divide), %% (remainder). Integer division truncates toward zero. Division by zero panics for integers.

Precedence level 5: additive: + (add), - (subtract). Overflow: panics in debug mode, wraps in release mode. Wrapping operations: wrapping_add, wrapping_sub.

Precedence level 6: shift: << (left shift), >> (right shift). Left shift fills with zeros. Right shift is: arithmetic for signed (sign extension), logical for unsigned (zero fill).

Precedence level 7: bitwise AND: &. Bitwise AND is used for: bit masking, flag testing. a & b extracts bits where both a and b have 1.

Precedence level 8: bitwise XOR: ^. XOR: 0^0=0, 0^1=1, 1^0=1, 1^1=0. XOR is used for: toggling bits, simple encryption, swap without temporary.

Precedence level 9: bitwise OR: |. OR: 0|0=0, 0|1=1, 1|0=1, 1|1=1. OR is used for: setting bits, combining flags.

Precedence levels 10-14: comparison (== != < > <= >=), logical AND (&&), logical OR (||), range (.. ..=), assignment (= += -= *= /= %%= <<= >>= &= ^= |=).

A.2 Appendix: Keyword Reference

Declaration keywords: fn (function), struct (structure), enum (enumeration), trait (trait), impl (implementation), mod (module), use (import), type (alias), const (constant), static (static variable).

Control flow keywords: if, else, match, for, while, loop, break, continue, return. Control flow keywords direct execution order.

Type keywords: Self (current type), self (current instance), super (parent module), crate (current crate). Special identifiers in type context.

Safety keywords: unsafe (unsafe block/function), extern (external ABI). Safety keywords mark code regions with different safety guarantees.

Async keywords: async (asynchronous function/block), await (suspend until ready). Async transforms functions into state machines returning Future.

Binding keywords: let (variable binding), mut (mutable), ref (reference pattern), move (move closure). Binding keywords control variable properties.

Visibility keywords: pub (public), pub(crate) (crate-visible), pub(super) (parent-visible), pub(in path) (path-visible). Default visibility is private.

Trait keywords: where (constraint clause), dyn (trait object), impl (impl Trait). Keywords for specifying and using trait bounds.

Memory keywords: box (heap allocation - unstable), in (placement - unstable). Memory keywords control allocation.

Special keywords: as (type cast, import rename), _ (wildcard pattern), true/false (boolean literals). Miscellaneous keywords with specific uses.

A.3 Appendix: Standard Traits Reference

Formatting traits: Display (user-facing output via {}), Debug (developer output via {:?}), Binary, Octal, LowerHex, UpperHex (number formatting).

Comparison traits: PartialEq (== !=), Eq (total equality), PartialOrd (< > <= >=), Ord (total ordering). Eq requires reflexivity. Ord requires totality.

Conversion traits: From/Into (infallible conversion), TryFrom/TryInto (fallible conversion). From implies Into via blanket impl.

Iterator traits: Iterator (next()), IntoIterator (into_iter()), FromIterator (collect()), ExactSizeIterator (len()), DoubleEndedIterator (next_back()).

Operator traits: Add, Sub, Mul, Div, Rem, Neg. Index, IndexMut. Deref, DerefMut. Not, BitAnd, BitOr, BitXor, Shl, Shr.

Memory traits: Clone (explicit copy), Copy (implicit copy), Drop (destructor), Default (default value). Copy implies Clone. Drop and Copy are mutually exclusive.

Marker traits: Send (safe to transfer between threads), Sync (safe to share between threads via reference). Most types are Send + Sync.

IO traits: Read (byte input), Write (byte output), BufRead (buffered input), Seek (random access). IO traits work with: files, networks, buffers.

Async traits: Future (async computation), Stream (async iterator), AsyncRead (async byte input), AsyncWrite (async byte output).

Allocation traits: Allocator (custom memory allocator), GlobalAlloc (global allocator interface). Custom allocators enable: arena allocation, pool allocation.

A.4 Appendix: Compiler Phases

Phase 1 - Lexing: source text to token stream. The lexer handles: keywords, identifiers, literals, operators, comments, whitespace. Output: Vec[Token].

Phase 2 - Parsing: token stream to AST. The parser builds the abstract syntax tree. Error recovery: sync on semicolons and braces. Output: AST.

Phase 3 - Name resolution: resolve identifiers to definitions. Build: scope tree, import resolution, macro expansion. Output: resolved AST with DefIds.

Phase 4 - Type checking: infer and verify types. Hindley-Milner inference with: trait bounds, lifetime annotations, coercions. Output: typed AST.

Phase 5 - Borrow checking: verify ownership and lifetime rules. The borrow checker uses: control flow graph, liveness analysis, region inference. Output: validated AST.

Phase 6 - MIR lowering: AST to Mid-level IR. MIR is: a control flow graph, SSA form, explicit drops. MIR enables: optimization, borrow checking.

Phase 7 - MIR optimization: constant propagation, dead code elimination, inlining, move elimination. MIR passes: ~30 optimization passes.

Phase 8 - Codegen: MIR to target code. Backends: LLVM IR (primary), Cranelift (debug builds), GCC (gccrs). Output: object files.

Phase 9 - Linking: object files to executable. Linkers: system linker (cc), LLD (bundled), mold (fast). Link-time optimization (LTO): thin LTO, full LTO.

Phase 10 - Incremental compilation: cache intermediate results. Changed functions are recompiled. Unchanged functions use cached results. Reduces rebuild times.

5.3 Assignment Statements

Simple assignment: x = value;. The left-hand side must be a place expression (variable, field, index, dereference). The right-hand side is evaluated and stored.

Compound assignment: x += 1; x -= 1; x *= 2; x /= 2; x %%= 2;. Compound assignment combines: read, operate, and store into one statement.

Destructuring assignment: (a, b) = (1, 2);. Tuple destructuring assigns to multiple variables. Struct destructuring: Point { x, y } = point;.

Swap: std::mem::swap(&mut a, &mut b). Swaps two values without a temporary. Pointer swap: std::ptr::swap(ptr_a, ptr_b) swaps memory at pointers.

Replace: std::mem::replace(&mut dest, src). Stores src in dest and returns the old value. Used for: taking values out of mutable references.

Take: std::mem::take(&mut value). Replaces value with Default::default() and returns the old value. Requires: T: Default.

Assignment and ownership: assignment moves the value unless T: Copy. After move: the source is uninitialized. After copy: both source and destination are valid.

Overflow-checked assignment: x = x.checked_add(1).expect('overflow'). Checked operations return Option[T]. Saturating: x = x.saturating_add(1). Wrapping: x = x.wrapping_add(1).

Bitwise compound assignment: x &= mask; x |= flag; x ^= toggle; x <<= shift; x >>= shift. Bitwise assignments modify individual bits.

Assignment in unsafe: assigning to *ptr is unsafe. ptr::write(ptr, value) writes without reading old value. ptr::write_volatile(ptr, value) for hardware registers.

8.3 Exhaustiveness Checking

Exhaustive matching: the compiler verifies that match arms cover all possible values. Missing patterns produce compile errors with examples of uncovered values.

Boolean exhaustiveness: match b { true => ..., false => ... }. Both values must be covered. Missing one produces: non-exhaustive patterns: 'false' not covered.

Integer exhaustiveness: integers have too many values for exhaustive listing. Use: wildcard (_) or range patterns to cover remaining values.

Enum exhaustiveness: match shape { Circle(r) => ..., Square(s) => ..., Triangle(a, b, c) => ... }. All variants must be covered.

Non-exhaustive attribute: #[non_exhaustive] enum Error { A, B }. External code must include a wildcard arm. Allows adding variants without breaking changes.

Nested exhaustiveness: match pair { (Some(x), Some(y)) => ..., (Some(_), None) => ..., (None, Some(_)) => ..., (None, None) => ... }. All combinations checked.

Guard exhaustiveness: guards are not considered for exhaustiveness. match x { n if n > 0 => ... } is not exhaustive even though mathematically it covers all cases.

Reference exhaustiveness: match &value { &Pattern => ... } matches through references. The compiler sees through references for exhaustiveness checking.

Slice exhaustiveness: [first, rest @ ..] covers non-empty slices. [] covers empty slices. Together they are exhaustive for any slice.

Usefulness checking: the compiler also warns about unreachable patterns. Unreachable patterns are dead code. Warning: unreachable pattern.

10.3 Monomorphization Detail

Monomorphization: for each concrete type used with a generic function, the compiler generates a specialized copy. fn add[T: Add](a: T, b: T) -> T generates: add_i32, add_f64, etc.

Code bloat: monomorphization can increase binary size. Each specialization is a separate function in the binary. Mitigation: trait objects (dynamic dispatch).

Inline specialization: small generic functions are inlined after monomorphization. Inlining eliminates: function call overhead and enables: further optimization.

Cross-crate monomorphization: generic functions from dependencies are monomorphized in the user crate. The dependency provides: MIR for generic functions.

Compile time impact: monomorphization increases compile time proportional to: number of specializations * function complexity. Mitigation: reduce generic surface area.

Type erasure alternative: Box[dyn Trait] uses: vtable lookup, heap allocation. Benefits: single copy of the function, smaller binary. Cost: dynamic dispatch overhead.

Specialization (unstable): impl[T] Trait for T { default fn method() { generic_impl() } } impl Trait for String { fn method() { optimized_impl() } }. Specialized impls override defaults.

Layout monomorphization: generic structs are monomorphized to concrete layouts. Vec[i32] has: pointer to i32 array. Vec[String] has: pointer to String array. Different sizes and alignments.

Const generics monomorphization: [T; N] is monomorphized for each N. Array[i32, 3] and Array[i32, 4] are different types with different sizes.

Monomorphization and linking: monomorphized functions may be duplicated across compilation units. The linker deduplicates identical functions (ICF: identical code folding).

3.4 Type Coercions

Implicit coercions: the compiler inserts coercions at specific points. Coercion sites: let bindings, function arguments, return values, struct fields.

Deref coercion: &String to &str, &Vec[T] to &[T], &Box[T] to &T. Deref coercion chains: the compiler applies Deref repeatedly until the target type matches.

Reference coercion: &mut T to &T (reborrow). Mutable references can be used where immutable references are expected. This is always safe.

Unsizing coercion: [T; N] to [T] (array to slice), Box[Struct] to Box[dyn Trait] (concrete to trait object). Unsizing adds metadata (length or vtable).

Pointer coercion: &T to *const T, &mut T to *mut T. Reference to raw pointer conversion is safe. The reverse (raw to reference) requires unsafe.

Numeric coercion: no implicit numeric coercions in Lateralus. Use: as for explicit casting. i32 as f64 is explicit. This prevents: silent precision loss.

Never type coercion: ! (never type) coerces to any type. return, break, continue, panic!, and loop {} have type !. This allows: let x: i32 = return;

Function pointer coercion: fn(i32) -> i32 coerces to fn pointer. Closures that capture nothing coerce to function pointers. Named functions coerce to fn pointers.

Subtype coercion: &'static str coerces to &'a str for any 'a. Longer lifetimes coerce to shorter lifetimes. This is lifetime subtyping.

Trait object coercion: &dyn Trait + 'static coerces to &dyn Trait + 'a. Trait object lifetimes follow the same subtyping rules as references.

A.5 Appendix: Error Messages

E0001 - Unreachable pattern: a pattern in a match expression that can never be reached. The compiler proves the pattern is subsumed by earlier arms.

E0015 - Cannot call non-const function in const context: const evaluation only supports a subset of operations. Non-const functions produce this error.

E0106 - Missing lifetime specifier: a reference in a function signature needs a lifetime annotation. The elision rules could not determine the lifetime.

E0277 - Trait not implemented: T does not implement Trait. This occurs when a type is used where a trait bound is required but the type lacks the implementation.

E0308 - Mismatched types: expected Type1, found Type2. The compiler expected one type but found another. Common cause: wrong return type, wrong argument type.

E0382 - Use of moved value: a value was moved and then used. Solutions: clone before move, use references, or restructure to avoid the move.

E0425 - Cannot find value in scope: an identifier was used but not defined. Common causes: typo in variable name, missing import, out-of-scope variable.

E0499 - Cannot borrow as mutable more than once: two mutable borrows of the same value exist simultaneously. The borrow checker prevents this data race.

E0502 - Cannot borrow as mutable because it is also borrowed as immutable: mixing mutable and immutable borrows violates aliasing rules.

E0597 - Value does not live long enough: a reference outlives the value it points to. The value is dropped before the reference expires. Solutions: extend lifetime, clone, or restructure.

A.6 Appendix: Compiler Flags

Optimization levels: -C opt-level=0 (no optimization), -C opt-level=1 (basic), -C opt-level=2 (full), -C opt-level=3 (aggressive), -C opt-level=s (size), -C opt-level=z (minimal size).

Debug info: -C debuginfo=0 (none), -C debuginfo=1 (line tables only), -C debuginfo=2 (full). Debug info enables: debugger breakpoints, stack traces, variable inspection.

Link-time optimization: -C lto=thin (fast LTO), -C lto=fat (full LTO), -C lto=off (no LTO). LTO enables: cross-crate inlining and optimization.

Target specification: --target x86_64-unknown-linux-gnu (Linux 64-bit), --target aarch64-apple-darwin (macOS ARM), --target wasm32-unknown-unknown (WebAssembly).

Code generation: -C codegen-units=1 (single CGU for best optimization), -C codegen-units=16 (parallel CGU for faster compilation). Trade-off: compile speed vs runtime performance.

Panic strategy: -C panic=unwind (default, stack unwinding), -C panic=abort (immediate abort). Abort: smaller binary, no unwinding overhead, no catch_unwind.

CPU features: -C target-cpu=native (use all CPU features), -C target-feature=+avx2,+fma (enable specific features). CPU features enable: SIMD instructions.

Linker selection: -C linker=clang (use clang as linker), -C link-arg=-fuse-ld=lld (use LLD), -C link-arg=-fuse-ld=mold (use mold for faster linking).

Profile-guided optimization: -C profile-generate (instrument binary), -C profile-use=profile.profdata (use profile for optimization). PGO improves: branch prediction, inlining.

Incremental compilation: -C incremental=target/incremental (cache directory). Incremental compilation caches: MIR, monomorphized items, codegen units.

A.7 Appendix: Memory Layout

Primitive layout: bool (1 byte), i8/u8 (1 byte), i16/u16 (2 bytes), i32/u32/f32 (4 bytes), i64/u64/f64 (8 bytes), i128/u128 (16 bytes), usize/isize (pointer-sized).

Struct layout (default): fields are reordered for minimal padding. The compiler may place smaller fields in padding gaps. Size is rounded up to alignment.

Struct layout (repr C): fields are laid out in declaration order. Padding follows C rules. Compatible with C struct layout. Required for: FFI.

Enum layout: tag + union of variants. Tag size: u8 for < 256 variants. Niche optimization: Option[&T] has no tag (null represents None). Size equals: tag + largest variant.

Slice layout: &[T] is (pointer, length). Fat pointer: 2 * pointer size. The pointer points to the first element. The length is the number of elements.

Trait object layout: &dyn Trait is (data pointer, vtable pointer). Fat pointer: 2 * pointer size. The vtable contains: drop, size, alignment, and trait method pointers.

Vec layout: Vec[T] is (pointer, length, capacity). Three usize fields. Total: 3 * pointer size on stack. Data is heap-allocated.

String layout: String is (pointer, length, capacity). Same as Vec[u8] with UTF-8 guarantee. &str is (pointer, length). Same as &[u8] with UTF-8 guarantee.

Box layout: Box[T] is a single pointer. No additional metadata. The pointer is non-null and uniquely owned. Size: pointer size.

Zero-sized types: () and PhantomData[T] have size 0. ZSTs do not occupy memory. Vec[()]: length is tracked but no heap allocation for data. ZSTs are useful as markers.

A.8 Appendix: Unsafe Code Checklist

Raw pointer dereference: verify pointer is non-null, aligned, points to initialized memory of correct type, and respects aliasing (no active &mut or &).

Union field access: verify the union currently stores the accessed variant. Document the invariant that ensures correctness. Consider using enum instead.

Mutable static access: verify no data races (use atomics or locks). Consider using: thread_local!, once_cell::Lazy, or atomic types instead.

Unsafe function call: read and satisfy all documented preconditions. Verify: pointer validity, alignment, initialization, and lifetime requirements.

FFI calls: verify: correct ABI, correct argument types, correct return type, null-termination of strings, ownership transfer rules.

Implementing unsafe traits: verify all safety invariants documented by the trait. Send: the type is safe to transfer. Sync: the type is safe to share.

Inline assembly: verify: correct register constraints, memory clobbers, side effects. Test on all target platforms.

Transmute: verify: same size, valid bit patterns for target type, alignment compatibility. Consider: from_bits, to_bits, or union instead.

Custom allocator: verify: correct alignment, correct size, no overlap with other allocations, proper deallocation of all allocated memory.

Pin violation: verify: pinned values are not moved. Drop implementation must not move the pinned value. Structural pinning requires careful analysis.

Testing unsafe code: run with: Miri (UB detection), AddressSanitizer (memory errors), ThreadSanitizer (data races), and extensive fuzzing. Document all safety proofs.

Unsafe code review: require: safety comments on every unsafe block, peer review, and automated checking tools. Minimize the amount of unsafe code.

A.9 Appendix: Edition Differences

Edition 2015: original edition. Path syntax: extern crate required. Trait objects: dyn keyword optional. Module system: mod.rs required for modules.

Edition 2018: path changes: no extern crate needed (except for macros). dyn keyword required for trait objects. Module system: filename modules (foo.rs = mod foo).

Edition 2021: closure capture changes: closures capture individual fields, not entire variables. Array IntoIterator: for x in array iterates elements directly.

Edition 2024: planned changes: lifetime capture rules, keyword reservations, and pattern matching updates. Edition migration tool: lateralus fix --edition 2024.

Cross-edition compatibility: crates of different editions can interoperate. The compiler translates between editions at crate boundaries. Public API is edition-independent.

Edition migration: lateralus fix --edition 2021 automatically applies most changes. Manual review needed for: semantic changes, ambiguous cases, and unsafe code.

Feature flags: unstable features require: #![feature(name)] and nightly compiler. Feature flags are removed when the feature stabilizes.

Stability guarantees: stable Lateralus: backward compatible. Code that compiles today will compile in the future. Stability applies to: language syntax, standard library API, compiler behavior.

Deprecation: deprecated features produce warnings. Deprecated features remain functional. Removal requires: edition boundary and migration path.

Migration testing: CI should test on: current stable, next edition preview, and nightly. This ensures: timely detection of upcoming changes.

A.10 Appendix: Concurrency Patterns

Message passing: std::sync::mpsc provides multi-producer, single-consumer channels. tx.send(value) sends. rx.recv() receives. Channels transfer ownership.

Shared state: Arc[Mutex[T]] provides thread-safe shared mutable state. Arc provides shared ownership. Mutex provides mutual exclusion. Lock contention is the bottleneck.

Read-write lock: RwLock[T] allows multiple readers or one writer. Readers do not block each other. Writers block all. Good for read-heavy workloads.

Lock-free patterns: AtomicBool for flags, AtomicUsize for counters, AtomicPtr for pointers. Lock-free algorithms avoid: deadlocks, priority inversion, convoying.

Thread pool: rayon::ThreadPool provides work-stealing parallelism. par_iter() parallelizes iterators. join() runs two closures in parallel. install() scopes the pool.

Scoped threads: std::thread::scope(|s| { s.spawn(|| ...) }) guarantees all threads finish before the scope ends. Scoped threads can borrow local variables.

Barrier: Barrier::new(n) synchronizes n threads. barrier.wait() blocks until all threads reach the barrier. Used for: phased computation, parallel initialization.

Condvar: Condvar with Mutex enables: wait-until-condition patterns. condvar.wait(guard) releases the lock and waits. condvar.notify_one() wakes one waiter.

Once: std::sync::Once::call_once(|| init()). Guaranteed to run exactly once across all threads. Used for: one-time initialization of global state.

Async concurrency: tokio::spawn(async { ... }) runs an async task. select! waits for the first of multiple futures. join! waits for all futures.

Deadlock prevention: acquire locks in a consistent order. Use try_lock() with timeout. Prefer channels over shared state. Use lock hierarchies.

Thread safety patterns: Send means safe to move between threads. Sync means safe to share references between threads. Rc is !Send + !Sync. Arc is Send + Sync.

A.11 Appendix: Testing Framework

Unit tests: #[test] fn test_name() { assert_eq!(actual, expected); }. Unit tests live in the same file as the code. Run with: lateralus test.

Assert macros: assert!(condition), assert_eq!(left, right), assert_ne!(left, right). Debug output on failure. Custom messages: assert!(cond, 'msg: {}', detail).

Integration tests: files in tests/ directory. Each file is a separate crate. Use: use my_crate::public_api;. Integration tests test the public API.

Doc tests: code blocks in /// comments are compiled and run. fn main() is implicit. assert_eq! verifies examples. Doc tests ensure: documentation accuracy.

Test organization: #[cfg(test)] mod tests { use super::*; }. Test modules are only compiled during testing. Private functions are accessible in test modules.

Test attributes: #[ignore] skips a test. #[should_panic] expects a panic. #[should_panic(expected = 'message')] matches panic message.

Property testing: proptest crate generates random inputs. proptest!(|(x in 0..100i32)| { assert!(f(x) >= 0); }). Shrinks failing inputs to minimal counterexamples.

Benchmark tests: criterion crate for statistical benchmarks. b.iter(|| expensive_function()). Criterion: detects performance regressions, generates reports.

Mocking: mockall crate auto-generates mock implementations from traits. #[automock] trait Service { fn get(&self) -> i32; }. MockService::new().expect_get().return_const(42).

Code coverage: source-based coverage with -C instrument-coverage. Generate report with: grcov. Coverage shows: which lines are tested.

Fuzzing: cargo-fuzz with libfuzzer. fuzz_target!(|data: &[u8]| { process(data); }). Fuzzing discovers: crashes, panics, and undefined behavior in unsafe code.

Snapshot testing: insta crate captures output and compares to stored snapshots. insta::assert_snapshot!(output). Review changes with: cargo insta review.

A.12 Appendix: Toolchain and Ecosystem

Package manager: lateralus-pkg manages dependencies. lateralus-pkg.toml declares: dependencies, features, targets, and metadata. lateralus-pkg.lock pins exact versions.

Build system: lateralus-pkg build compiles the project. Build scripts: build.rs runs before compilation. Build scripts generate: code, link native libraries, set cfg flags.

Formatter: lateralusfmt formats code. Configuration: lateralusfmt.toml. Default style: 4-space indentation, trailing commas, sorted imports.

Linter: lateralus-clippy provides 600+ lints. Categories: correctness, style, complexity, performance, pedantic. #[allow(clippy::lint_name)] suppresses specific lints.

Documentation: lateralusdoc generates HTML documentation. Features: search, source links, examples, trait implementation lists. Publish to: docs.rs.

Cross-compilation: lateralus-pkg build --target aarch64-unknown-linux-gnu. Required: target-specific linker and sysroot. Cross crate simplifies: cross-compilation setup.

Release profiles: [profile.release] opt-level = 3, lto = true, codegen-units = 1, strip = true. Profiles customize: optimization, debug info, and binary size.

Workspace management: [workspace] members = ['crate1', 'crate2']. Workspaces share: Cargo.lock, target directory, and dependency resolution. Monorepo support.

Feature flags: [features] default = ['std'], async = ['tokio']. Features enable conditional compilation. Feature unification: union of all requested features.

Publishing: lateralus-pkg publish uploads to the registry. Pre-publish checks: version bump, changelog, license, documentation. Yanking: lateralus-pkg yank --version removes a version.

A.13 Appendix: Performance Optimization

Profiling: perf record ./program, perf report (Linux). Instruments (macOS). flamegraph crate generates SVG flame graphs from perf data.

Allocation profiling: DHAT tracks: allocations, deallocations, and peak memory. jemalloc-ctl provides runtime statistics. heaptrack visualizes allocation patterns.

Iterator optimization: iterators compile to: loop with direct element access. No heap allocation. No virtual dispatch. Equivalent to hand-written loops.

SIMD: std::arch::x86_64 provides intrinsics. Portable SIMD (std::simd) provides cross-platform vector types. Auto-vectorization: the compiler may vectorize loops.

Cache optimization: data-oriented design keeps related data contiguous. SoA (struct of arrays) vs AoS (array of structs). Cache-friendly access patterns: sequential, predictable.

Inlining hints: #[inline] suggests inlining. #[inline(always)] forces inlining. #[inline(never)] prevents inlining. Small functions are inlined automatically.

String optimization: use &str instead of String where possible. Small string optimization: compact_str stores short strings inline. Avoid: repeated allocation with format!.

Collection preallocation: Vec::with_capacity(n), HashMap::with_capacity(n). Preallocating avoids: repeated reallocation and copying during growth.

Zero-copy parsing: nom and winnow parse without allocating. Parsers return: references into the input buffer. Zero-copy is critical for: high-throughput data processing.

Async performance: avoid: holding locks across await points, spawning too many tasks, blocking in async context. Use: tokio::task::spawn_blocking for CPU-heavy work.

Compile-time computation: const fn evaluates at compile time. Build scripts precompute data. Proc macros generate optimized code. Compile-time computation eliminates: runtime overhead.

Memory pool allocation: arena allocators (bumpalo) provide fast allocation with bulk deallocation. Object pools reuse allocations. Pool allocation reduces: allocator pressure.

A.14 Appendix: Platform-Specific Behavior

Conditional compilation: #[cfg(target_os = 'linux')] fn platform_init() { }. cfg attributes select code for specific platforms. cfg_if! macro provides if-else syntax.

Target triples: arch-vendor-os-env. Examples: x86_64-unknown-linux-gnu, aarch64-apple-darwin, wasm32-wasi. Target triples specify: CPU, vendor, OS, and ABI.

Pointer width: cfg(target_pointer_width = '64') selects 64-bit platforms. usize and isize are pointer-width integers. Address space differs: 32-bit (4 GB), 64-bit (16 EB).

Endianness: cfg(target_endian = 'little') for little-endian. x86, ARM (default): little-endian. Network byte order: big-endian. Use: to_le_bytes(), to_be_bytes() for conversion.

OS-specific APIs: std::os::unix, std::os::windows provide platform-specific extensions. Unix: file permissions (mode), symlinks, signals. Windows: handles, wide strings.

ABI compatibility: extern 'C' uses the C calling convention. extern 'system' uses the OS calling convention (stdcall on Windows 32-bit, C elsewhere).

Feature detection: is_x86_feature_detected!('avx2') checks CPU features at runtime. Runtime detection enables: SIMD dispatch to optimal implementation.

Atomic availability: not all platforms support all atomic sizes. cfg(target_has_atomic = '64') checks availability. Fallback: use Mutex for platforms without atomics.

Path separators: std::path handles platform differences automatically. / on Unix, \ on Windows. Path::new() normalizes separators. Use Path, never string manipulation.

Line endings: text mode converts line endings on Windows (CRLF to LF). Binary mode preserves bytes. Use: BufReader for text, read_exact for binary.

Stack size: default thread stack size varies by platform (8 MB on Linux, 1 MB on Windows). thread::Builder::stack_size(bytes) customizes.

Signal handling: Unix signals (SIGINT, SIGTERM) require: signal-hook crate or ctrlc crate. Windows uses: SetConsoleCtrlHandler. Async signals: tokio::signal.

A.15 Appendix: Security Considerations

Memory safety: the ownership system prevents: use-after-free, double-free, buffer overflow, dangling pointers. Safe Lateralus has no undefined behavior.

Integer overflow: debug mode panics on overflow. Release mode wraps. Security-sensitive code should use: checked_add, saturating_add, or the overflow-checks profile option.

Input validation: validate all external input. Use: type-safe parsers (serde), bounded types (NonZeroU32), and validation libraries. Never trust user input.

Cryptographic safety: use: well-audited crates (ring, rustls). Avoid: implementing custom cryptography. Use: constant-time comparison for secrets. Clear secrets from memory.

Supply chain security: cargo-vet reviews dependency changes. cargo-crev provides community reviews. cargo-audit checks for: known vulnerabilities. Pin dependencies.

Sandboxing: seccomp (Linux), pledge (OpenBSD) restrict system calls. Capability-based security limits: file access, network access. Reduce the attack surface.

Unsafe audit: minimize unsafe code. Document safety invariants. Test with Miri. Use: #![forbid(unsafe_code)] in crates that should be entirely safe.

Timing attacks: constant-time operations prevent: timing-based information leaks. subtle crate provides: constant-time comparison, conditional selection.

Fuzzing for security: cargo-fuzz discovers: crashes, panics, and logic errors. afl-rs provides: American Fuzzy Lop integration. Continuous fuzzing with: OSS-Fuzz.

Dependency minimization: fewer dependencies means: smaller attack surface, less supply chain risk, faster audits. Prefer: standard library over third-party crates when possible.

Secrets management: zeroize crate clears sensitive data from memory on drop. secrecy crate wraps secrets with: zeroize-on-drop and Debug redaction.

Hardening flags: -C relro-level=full (read-only relocations), -C stack-protector=all (stack canaries), ASLR (address space layout randomization, OS-level).

A.16 Appendix: Future Language Directions

Async traits: currently unstable. Async fn in traits enables: trait-based async interfaces without boxing. Expected stabilization: planned for upcoming edition.

Generators: yield-based generators produce iterators from imperative code. gen fn fibonacci() -> i32 { let (a, b) = (0, 1); loop { yield a; (a, b) = (b, a + b); } }

Specialization: default impl methods can be overridden by more specific implementations. Enables: efficient implementations for specific types while maintaining generality.

Generic associated types (GATs): type Item['a] in traits. GATs enable: lending iterators, streaming iterators, and more expressive trait definitions. Stabilized recently.

Const generics: expanding from [T; N] to support: const expressions, const trait bounds, and const fn in generic context. Partial stabilization ongoing.

Allocator API: custom allocators as type parameters. Vec[T, A: Allocator]. Enables: arena allocation, stack allocation, and specialized memory management per container.

Pattern types: types constrained by patterns. type NonZero = i32 is 1..;. Pattern types enable: zero-cost range constraints, non-null guarantees, and enum optimization.

Effect system: potential future feature. Effects: async, try, const, unsafe as composable annotations. Effect polymorphism: generic over presence of effects.

Variadic generics: functions accepting variable numbers of type parameters. Enables: tuple manipulation, heterogeneous containers, and type-level lists.

Linear types: types that must be used exactly once. Stronger than affine types (which can be used at most once). Enables: guaranteed resource cleanup, protocol enforcement.

Placement new: allocating directly into the target location without intermediate moves. Enables: efficient construction of large objects and container elements.

Context and capabilities: implicit parameters passed through the call stack. Enables: dependency injection, logging context, and resource access without global state.

References

[1] Pierce, B. C. Types and Programming Languages. MIT Press, 2002.

[2] Stroustrup, B. The Design and Evolution of C++. Addison-Wesley, 1994.

[3] Milner, R. et al. The Definition of Standard ML. MIT Press, 1997.

[4] Jones, S. P. et al. The Haskell 98 Language Report. JFP, 2003.

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

[6] Lattner, C. and Adve, V. LLVM: A Compilation Framework. CGO, 2004.