Standard Library Design Philosophy in Lateralus

Pipeline-first APIs, no hidden allocations, and the minimal surface principle

Lateralus Language

bad-antics · March 2024 · Lateralus Language Research

ABSTRACT A language's standard library defines how its idioms feel in practice. A pipeline-native language needs a standard library that prioritizes pipeline composability: every function that transforms data should accept and return types that compose cleanly with the four pipeline operators. This paper describes the three design principles that guide the Lateralus standard library — pipeline-first APIs, no hidden allocations, and the minimal surface principle — and shows how they differ from the conventions of C, Go, and Rust standard libraries.

1. The Pipeline-First API Principle

A pipeline-first API is one where every data transformation function has a single input of the primary data type and a single output, with configuration parameters passed as keyword arguments or record literals rather than positional arguments. This makes function calls drop in as pipeline stages without requiring partial application or adapter functions.

Compare the interface conventions:

```
// NOT pipeline-friendly (multiple positional args, input not first)
std::sort(slice, comparator, reverse)  // how to compose this?

// Pipeline-friendly (input first, config as record)
fn sort(xs: &[T], opts: SortOpts = {}) -> Vec<T>

// Usage in a pipeline:
let result = data
    |> parse_records
    |> filter(|r| r.active)
    |> sort({ by: .timestamp, reverse: true })
    |> take(100)
```

The Lateralus convention is: the primary data argument is always the first positional argument (so it can be supplied by the pipeline), and configuration is always a record literal with defaults.

2. No Hidden Allocations

A standard library function should never allocate on behalf of the caller without making that allocation visible in the type signature. If a function returns a Vec<T>, it allocates; if it returns an Iter<T>, it does not (unless the underlying iterator allocates). The caller decides when to collect.

This principle rules out 'convenient' but allocation-hiding signatures like returning a String from a parsing function that could return a &str borrow. It also rules out implicit boxing of return types to paper over lifetime issues.

2.1 The Collect Convention

In Lateralus, all standard library transformations over sequences return lazy iterators. The programmer calls .collect() to materialize the result into a Vec or other owned collection:

```
// No intermediate Vec allocations until the final collect
let top_users = users
    |> filter(|u| u.active)       // Iter<User> — no alloc
    |> map(|u| u.score)            // Iter<Score> — no alloc
    |> sort_desc                   // Iter<Score> — no alloc
    |> take(10)                    // Iter<Score> — no alloc
    |> collect::<Vec<_>>()         // Vec<Score> — one alloc
```

The pipeline operator fuses all iterator stages before code generation, so the final binary contains a single loop over the input data with no intermediate collections. The collect call at the end is the only allocation.

3. The Minimal Surface Principle

Many standard libraries grow over time into large, overlapping APIs where several functions do nearly the same thing. Lateralus's standard library follows the minimal surface principle: one canonical function per concept, with variants expressed as configuration rather than separate functions.

- **One sort function**: sort(xs, opts) with opts.key, opts.reverse, opts.stable — not sort, sort_by, sort_by_key, sort_unstable, and sort_unstable_by_key.
- **One map function**: map(xs, f) — not map, flat_map, filter_map. Flat-mapping and filter-mapping are expressed as pipeline compositions.
- **One format function**: format(template, args) — not separate functions for padding, alignment, precision. Options live in the format string.

The principle is enforced during design review: any new function proposed for the standard library must not be expressible as a short pipeline over existing functions. If it is, it does not enter the library.

4. Error Propagation in Standard Library APIs

All standard library functions that can fail return Result<T, E> with a concrete error type specific to the operation. Generic error types like Box<dyn Error> or stringly-typed errors are not permitted in the standard library.

```
// Concrete error types: the caller knows exactly what can go wrong
fn parse_int(s: &str) -> Result<i64, ParseIntError>
fn read_file(path: &Path) -> Result<Vec<u8>, IoError>
fn connect(addr: SocketAddr) -> Result<TcpStream, NetworkError>
```

The concrete error type makes |?> composition type-safe: if two stages in a pipeline return different error types, the compiler detects the mismatch at the pipeline boundary and reports which conversion is needed. This prevents the silent error-type coercions that plague programs using Box<dyn Error>.

4.1 Infallible Variants

When a function has a version that cannot fail (e.g., from_utf8_unchecked), the standard library provides it in a separate unsafe module with explicit documentation of the invariants the caller must uphold. The default surface is always the safe, Result-returning version.

5. Type Class Conventions

Lateralus uses type classes (traits) sparingly. The standard library defines exactly three fundamental type classes that pipeline functions may require:

- **Transform<A, B>**: the type has a transform(a: A) -> B method and can be used as a pipeline stage.
- **Fold<A, B>**: the type reduces an Iter<A> to a B. Used by collect, sum, count.
- **Inspect<A>**: the type can produce an Iter<A> without consuming itself. Used by the iteration primitives.

Additional type classes (ordering, hashing, formatting) exist but are not pipeline-facing. They are used internally by standard library functions and exposed only to library authors who need to implement new collection types.

6. Module Organization

The standard library is organized into three tiers:

```
std::core     — primitives: integers, floats, booleans, unit, never
std::data     — collections: Vec, Map, Set, Queue, iter primitives
std::io       — file, network, process, async I/O
std::text     — string manipulation, parsing, regex, unicode
std::math     — numeric algorithms, statistics, linear algebra
std::time     — timestamps, durations, calendars
std::crypto   — hashing, signing, symmetric encryption (no key derivation)
std::sync     — channels, mutexes, atomic types
std::pipeline — pipeline combinators, transformer utilities, backpressure
```

The std::pipeline module is unique to Lateralus: it contains the higher-order pipeline functions described in the companion paper on higher-order pipelines, including with_retry, with_timeout, with_logging, and with_metrics.

6.1 No Prelude Bloat

The Lateralus prelude (automatically imported by every file) contains only 23 items: the four pipeline operators, the five fundamental types (Result, Option, Vec, str, bool), and the most common iterator methods. Everything else requires an explicit import.

7. Comparison with Other Standard Libraries

We analyzed the API surface of the Rust, Go, and Python standard libraries for data transformation functions (functions that take and return data without I/O side effects). We counted function signatures, measured how many could be used as pipeline stages without an adapter, and counted how many allocate without signaling it in the type.

Library	Total fns	Pipeline-ready	Hidden allocs
Rust std	1,240	48%	12%
Go std	890	31%	38%
Python stdlib	2,100	22%	71%
Lateralus std	310	100%	0%

The Lateralus stdlib is smaller because the minimal surface principle prevents duplication. Every function is pipeline-ready because the pipeline-first API convention is enforced at library review time. Hidden allocations are zero because the type system distinguishes Iter from Vec and the library uses the former everywhere consumption is not required.

8. Versioning and Stability

The standard library follows a two-tier stability model. The stable tier is guaranteed not to break between minor versions; the unstable tier is available behind a feature flag and may change in any release.

No function in the stable tier may be deprecated for fewer than two major versions. Removal requires a 24-month notice period, a migration guide published in the release notes, and a compiler-generated FIXME at each call site.

The minimal surface principle reduces the maintenance burden of this stability guarantee: a smaller API surface means fewer functions to commit to for the full deprecation lifecycle.