# Polyglot Bridge Internals

Calling C, Python, and Rust from Lateralus without wrapper boilerplate

Lateralus Language

bad-antics · July 2024 · Lateralus Language Research

**ABSTRACT** Lateralus's polyglot bridge enables calling functions in C, Python, and Rust from Lateralus code with zero hand-written wrapper code. The bridge uses a three-layer architecture: a compile-time interface extractor that reads foreign headers or type stubs, a type-mapping layer that converts between Lateralus and foreign types, and a runtime ABI adaptor that handles calling convention differences. This paper describes each layer, explains the type-safety guarantees and their limits, and benchmarks the overhead versus hand-written FFI bindings.

## 1. Motivation: FFI Without Wrappers

Foreign Function Interfaces (FFI) in most languages require the programmer to write explicit binding code: a C header declaration in Rust, a ctypes structure in Python, or a foreign import declaration in Haskell. For large C libraries (libc, OpenSSL, SQLite), the binding code can exceed 10,000 lines and must be maintained in sync with the upstream headers.

Lateralus's polyglot bridge eliminates wrapper code for the common case: when the foreign function's types map cleanly to Lateralus types, the bridge generates the binding automatically from the foreign header or type stub. The programmer writes one import declaration:

```
// Import from a C library
import foreign::c { header: "<sqlite3.h>", lib: "sqlite3" }

// All sqlite3_* functions are now callable directly
let db = sqlite3_open("data.db")?
```

When types do not map cleanly (e.g., C unions, variadic functions, pointer-to-pointer patterns), the programmer writes a thin adapter in a separate file and the bridge wraps the adapter instead of the original function.

## 2. Layer 1: Interface Extraction

The interface extractor runs at compile time. For C headers, it uses a minimal C preprocessor and parser to extract function declarations, typedef'd struct definitions, and enum values. It does not evaluate preprocessor macros that expand to expressions (those require manual binding).

```
// Extracted from sqlite3.h (simplified)
foreign_fn sqlite3_open(filename: *const u8, db: **sqlite3) -> i32
foreign_fn sqlite3_close(db: *sqlite3) -> i32
foreign_fn sqlite3_exec(db: *sqlite3, sql: *const u8,
                        cb: *fn, arg: *void, errmsg: **u8) -> i32
```

For Python modules, the extractor reads .pyi stub files (PEP 484) and maps Python type annotations to Lateralus types. For Rust crates, it reads the crate's cbindgen.toml and generated C header to extract the public ABI.

### 2.1 Extraction Limits

The extractor handles: function pointers as callback types, null-terminated string conventions (*const u8 → CStr), fixed-size arrays, and opaque handle types. It does not handle: C++ classes, variadic arguments, setjmp/longjmp exception models, or platform-specific ABI extensions.

## 3. Layer 2: Type Mapping

The type mapper converts foreign types to Lateralus types and back. The mapping is injective on the safe subset: every safe Lateralus type has a unique foreign representation, but not every foreign type has a safe Lateralus equivalent.

```
Foreign Type         Lateralus Type      Safety
-------------------------------------------------
i8, i16, i32, i64   i8, i16, i32, i64   safe
u8, u16, u32, u64   u8, u16, u32, u64   safe
f32, f64            f32, f64             safe
bool (C _Bool)      bool                 safe
*const T            &T (borrow)          safe (no null)
*mut T              &mut T               safe (no null)
*const u8 (NUL-term) CStr                safe
struct { fields }   record { fields }    safe
void*               unsafe::RawPtr       unsafe
union { ... }       unsafe::ForeignUnion unsafe
int (*)(...)        unsafe::FnPtr        unsafe
```

When the mapper encounters an unsafe type, it wraps the foreign function in an unsafe block in the generated binding. The programmer must explicitly call the function within an unsafe { } scope, acknowledging the invariants they are responsible for maintaining.

## 4. Layer 3: ABI Adaptor

The ABI adaptor handles calling convention differences at runtime. On x86-64 Linux, C uses the System V AMD64 ABI; on Windows, C uses the Microsoft x64 ABI. Lateralus uses the System V ABI for its native calls and generates a thunk when calling Windows binaries on Linux or vice versa.

For Python, the bridge calls the CPython C API directly: arguments are converted from Lateralus values to PyObject* via the type mapping layer, the Python function is called via PyObject_Call, and the return value is converted back. The GIL is acquired before the call and released after if the call is marked blocking.

```
// Calling a Python function from Lateralus
import foreign::python { module: "numpy" }

let array = numpy::array([1.0, 2.0, 3.0, 4.0])
let mean = numpy::mean(array)  // returns f64
```

### 4.1 Error Convention Mapping

C functions signal errors via return codes; Python via exceptions; Rust via Result. The bridge maps each convention to Lateralus's Result. For C, the programmer annotates the expected success code in the import declaration; for Python, any raised exception is caught and converted to Err; for Rust, the Result is passed through directly.

## 5. Pipeline Integration

Foreign functions that pass through the type mapper become usable as pipeline stages without any additional adapter code. A C function process(data: *const u8, len: usize) -> i32 becomes a Lateralus function process(data: &[u8]) -> Result<i32, i32> and can be used in a pipeline:

```
let result = raw_bytes
    |> compress
    |?> process        // foreign C function, error-propagating
```

```
    |>  format_result
```

The bridge generates the type conversion code at each call site so that the pipeline form is ergonomic. The overhead of the conversion is measured in the benchmarks section.

## 6. Safety Guarantees and Their Limits

The bridge provides the following guarantees for the safe-type subset:

- No null pointer dereferences: safe pointer types in Lateralus are non-null by construction; the bridge verifies at the call site that no null is passed to a non-nullable foreign parameter.
- No buffer overflows: slice types carry length metadata; the bridge passes both the pointer and the length to foreign functions that expect separate pointer/length arguments.
- No use-after-free: the lifetime system ensures that borrows passed to foreign functions do not outlive the owning value.

The bridge does NOT guarantee memory safety for foreign code itself: if the C function has an internal buffer overflow, Lateralus cannot detect it. The bridge only guarantees that the Lateralus side of the call is safe.

## 7. Performance Benchmarks

We compared the polyglot bridge overhead against hand-written FFI bindings for three scenarios: calling a simple C math function, calling a struct-taking C library function, and calling a Python function via CPython API.

```
Scenario                  Bridge overhead vs hand-written FFI
-------------------------------------------------------------
C scalar function             < 1 ns      (zero overhead, inlined)
C struct-passing function     ~3 ns       (struct copy, unavoidable)
C callback registration       ~5 ns       (function pointer wrap)
Python function call          ~800 ns     (GIL + PyObject overhead)
```

The C overhead is negligible: the bridge generates the same code as a hand-written binding after inlining. The Python overhead is inherent to the CPython API and is no worse than calling from C via the same API.

## 8. Future Work

Planned improvements to the polyglot bridge: automatic binding generation for gRPC and Cap'n Proto schemas (treating RPC as a foreign call layer), support for WASM modules as a language-neutral foreign runtime, and a bridge-aware fuzzer that generates valid inputs for foreign functions based on their extracted type signatures.

The type-mapping layer will be extended to handle C++ templates via libclang integration, enabling direct calls into modern C++ libraries without an intermediate C wrapper layer.