The Lateralus C Backend

Transpiling a pipeline-first language to freestanding C99

Lateralus Language

bad-antics · April 2026 · Lateralus Compiler Internals

ABSTRACT The Lateralus compiler ships a C backend that emits either hosted or freestanding C99. The hosted mode is useful for embedding Lateralus in existing C projects; the freestanding mode is what lets LateralusOS's kernel integrate Lateralus-authored modules without a libc. This paper describes the transpiler's runtime library, the value representation, the lowering rules for pipelines and closures, and the escape hatches we provide for FFI-heavy code. We close with benchmarks comparing the C-backend output to the VM on the canonical example set.

1. Goals and Non-Goals

1.1 Goals

- **Readable output.** A developer should be able to open the generated .c file and follow it. No one-letter identifiers, no compressed whitespace, no line-noise.
- **Portable C99.** The output compiles under GCC 9+, Clang 10+, and MSVC 2019+. No compiler-specific extensions in the generated code; only in the optional runtime shims.
- **Freestanding optional.** A --freestanding flag produces code with no dependency on libc, malloc, stdio, or string.h. All such services are provided by the embedding environment.
- **Round-trippable semantics.** A Lateralus source file that passes the VM's test suite must also pass the same tests when compiled through the C backend and executed as native code.

1.2 Non-Goals

- **Minimum binary size.** We prioritize readable output over aggressive tree-shaking; users who want the latter can run strip and link-time DCE.
- **AOT compilation of the full language.** Reflection, eval, and dynamic module loading are not supported in the C backend; they remain VM-only.
- **Incremental compilation.** The backend recompiles the whole module on each invocation. Users wanting incremental builds can partition their code into multiple modules.

2. Value Representation

The runtime value is a tagged union:

```
typedef enum {
    LTL_INT, LTL_FLOAT, LTL_BOOL, LTL_STR,
    LTL_LIST, LTL_MAP, LTL_FN, LTL_NULL
} ltl_tag_t;

typedef struct ltl_val {
    ltl_tag_t tag;
    union {
        int64_t  i;
        double   f;
        int      b;
        ltl_str_t *s;
        ltl_list_t *l;
        ltl_map_t  *m;
        ltl_fn_t   *fn;
    } v;
} ltl_val_t;
```

Strings, lists, maps, and functions are heap-allocated and reference-counted. The ref-count is the first field of the target struct so that the ltl_retain and ltl_release macros do not need to know the specific type. On --freestanding the allocator is a user-provided ltl_alloc/ltl_free pair; in hosted mode they default to malloc/free.

3. Pipeline Lowering

The pipeline operator |> desugars at the AST stage into left-to-right function calls, so by the time the C emitter sees an expression there are no pipelines left: only Call nodes. This keeps the backend simple and pushes the non-obvious work into a single well-tested AST-lowering pass.

Concretely, xs |> map(f) |> filter(p) is equivalent at the AST level to filter(p, map(f, xs)). The backend emits:

```
ltl_val_t t1 = ltl_call2(map_builtin, f, xs);
ltl_val_t t2 = ltl_call2(filter_builtin, p, t1);
ltl_release(t1);                      // freed after use
return t2;
```

The ltl_release calls are inserted by a lifetime pass that runs after lowering but before emission; it walks each basic block in program order and emits a release after the last use of each temporary.

4. Closures

Closures in Lateralus capture by value (copies of bindings at closure-creation time). The C backend emits a closure as a pair of pointers: a function pointer and an environment struct pointer. The environment struct is generated per call site:

```
// source: let add = fn(x) { fn(y) { x + y } }
typedef struct { ltl_refcount_t rc; ltl_val_t x; } env_0_t;

static ltl_val_t lambda_inner(env_0_t *env, ltl_val_t y) {
    return ltl_add(env->x, y);
}

static ltl_val_t add_outer(ltl_val_t x) {
    env_0_t *env = ltl_alloc(sizeof(env_0_t));
    env->rc = 1;
    env->x  = ltl_retain(x);
    return ltl_make_fn(lambda_inner, env);
}
```

The environment struct is reference-counted; when the last closure holding it goes away, the struct and its captured values are released.

5. FFI

The @foreign annotation in Lateralus declares a function whose body is external C. The backend emits a declaration only:

```
@foreign("c", header="openssl/md5.h")
fn md5(data: bytes) -&gt; bytes
```

becomes:

```
#include &lt;openssl/md5.h&gt;
extern unsigned char *MD5(const unsigned char *, size_t, unsigned char *);
```

```
// wrapper inserted by runtime:
static ltl_val_t ltl_md5(ltl_val_t data) { ... }
```

For the hosted target, the linker resolves the external symbol normally. For the freestanding target, the user is expected to provide the implementation; the backend emits no assumptions beyond the declaration.

6. Runtime Library

The runtime is a single-file header-and-source pair:

- ltl_runtime.h — type definitions, macro declarations (~200 lines).
- ltl_runtime.c — reference-counting helpers, list/map implementations, string builder, pipeline built-ins (map, filter, reduce, zip, range), arithmetic dispatch, comparison, hashing (~1400 lines).

In freestanding mode the runtime is linked as a static object; the user's build system provides ltl_alloc, ltl_free, ltl_memcpy, and ltl_memset symbols. On LateralusOS these are backed by the kernel's heap allocator.

7. Freestanding Mode

The --freestanding flag toggles three behaviours:

- **No stdio.** println is declared as an extern that the user must supply.
- **No stdlib.** malloc/free symbols are replaced with ltl_alloc/ltl_free externs.
- **No string.h.** The runtime's own ltl_memcpy/ltl_memset/ltl_strlen are used.

The resulting object file can be linked with -ffreestanding -nostdlib flags and run in kernel context. LateralusOS uses this mode for the bootstrap utilities that were originally written in C but have since been ported to Lateralus: the command-line parser in the shell, the ELF loader, and the ramdisk walker.

8. Correctness Testing

The test strategy is a differential fuzzer: every file in examples/ and every file in tests/'s source corpus is compiled both through the VM and through the C backend, and the outputs are diffed. We run this nightly against the main branch. The diff has been clean for the last 38 consecutive nights as of this writing; the only divergences have been in examples that use reflection or eval, both of which the C backend correctly refuses with a named diagnostic.

9. Benchmarks

Wall-clock time for the canonical example set; hosted mode, GCC 13 with -O2:

- fibonacci.ltl (n=35): VM 52 ms, C backend 4 ms (13x).
- crypto_challenges.ltl: VM 850 ms, C backend 95 ms (9x).
- data_pipeline.ltl: VM 160 ms, C backend 22 ms (7x).
- physics_sim.ltl (10,000 frames): VM 2.9 s, C backend 280 ms (10x).

The speedups are roughly what you'd expect for a dynamic-language-to-C transpiler: native arithmetic, native function-call convention, no interpreter dispatch. The remaining overhead is in reference counting and in the tagged-value dispatch on arithmetic operations.

10. Future Work

- **Monomorphization pass.** For functions whose type is statically known, emit specialised C rather than going through the tagged-value path. Projected 2-4x further speedup on numeric code.
- **Region-based memory.** Replace reference counting with region allocators in the common fn ... { ... } case; cuts allocation churn on pipeline-heavy code.
- **WASM backend.** A parallel backend emitting WebAssembly using the same AST-lowering pass. Prototype in progress.

11. Conclusion

The C backend is a pragmatic balance: it's not the fastest possible Lateralus implementation, but it's readable, portable, and freestanding-capable, and it earns its place as the native deployment target for kernel and embedded use. The 10x speedup over the VM is comfortable, and the runtime fits in 1,600 lines of C including comments. Further optimisation work is planned but not on the critical path; the current backend is already the fastest Lateralus target and the one we recommend for production deployment.