

# *Enma*

## Style Guide

Conventions for writing Enma scripts, embedding the Enma SDK, and building single-player scripts on Perception.

```
int64 main() {  
    // Sized types. Clear ownership. No surprises.  
    println(std::to_string(Fib(30)));  
    return 0;  
}
```

# Inside this guide

|    |                                                   |    |
|----|---------------------------------------------------|----|
| 01 | Purpose and how to read this guide                | 2  |
| 02 | Baseline and toolchain                            | 4  |
| 03 | Project layout                                    | 5  |
| 04 | Naming                                            | 6  |
| 05 | Formatting and linting                            | 8  |
| 06 | Types and values                                  | 9  |
| 07 | Functions, lambdas and control flow               | 11 |
| 08 | Structs, classes and templates                    | 13 |
| 09 | Memory and ownership                              | 15 |
| 10 | Errors and exceptions                             | 16 |
| 11 | Modules, namespaces and the preprocessor          | 18 |
| 12 | Annotations and data layout                       | 20 |
| 13 | Embedding: engine, module and context lifecycle   | 22 |
| 14 | Native functions and addons                       | 24 |
| 15 | Threading and contexts                            | 26 |
| 16 | Safety: permissions, budgets and containment      | 27 |
| 17 | Performance                                       | 29 |
| 18 | Perception: scope, script structure and lifecycle | 30 |
| 19 | Perception: process access and permissions        | 32 |
| 20 | Perception: rendering and overlays                | 34 |
| 21 | Perception: API conventions and tooling           | 35 |
| 22 | Testing                                           | 36 |
| 23 | Git, releases and review                          | 37 |
| A  | Appendix                                          | 38 |

Rule levels

MUST

MUST NOT

SHOULD

SHOULD NOT

MAY

AVOID

# 01

## Purpose and how to read this guide

Enma is an embeddable, ahead-of-time compiled language with C++ semantics and script-grade containment: whole modules compile to native x64, with no bytecode and no interpreter, and scripts can reach only what the host registers. This guide covers both sides of an Enma project:

- **Script authors** writing `.em` modules.
- **Host authors** embedding the Enma SDK in a C++ application, registering natives and addons, and running scripts.

Enma is deliberately C++-faithful, so the [C++ Style Guide](#) applies to Enma scripts wherever the language allows it, and in full to host code. This guide adds what is specific to Enma: its sized types, its ownership and escape rules, its module system, and the rules for embedding it safely. Examples use a fictional host, `Project`, whose prefix is `PRJ`. Sections 18–21 cover Perception, the host these scripts are primarily written for.

### Rule levels

| LEVEL                                         | MEANING                                                        |
|-----------------------------------------------|----------------------------------------------------------------|
| <code>MUST</code> / <code>MUST NOT</code>     | Required. CI or review blocks the change.                      |
| <code>SHOULD</code> / <code>SHOULD NOT</code> | The default. Deviate only with a short comment explaining why. |
| <code>MAY</code>                              | Allowed when it helps; no justification needed.                |
| <code>AVOID</code>                            | Legal but discouraged; expect a review question.               |

### Principles

- **Write Enma, not C++.** Enma is C++-shaped but has its own spellings ( `int64`, not `long` ) and its own standard surface. Code that only compiles as C++ is a bug.
- **The host decides.** Scripts get exactly the natives, addons and permissions the host grants, and nothing else.
- **Deterministic ownership.** No garbage collector: every allocation has an owner and a scope.
- **Fail inside the script.** A script fault fails `execute()`; it never takes the host down.

- **Measure before tuning.** Enma is already native code; profile before reaching for annotations and intrinsics.

# 02

## Baseline and toolchain

| AREA          | STANDARD                                                                                                   |
|---------------|------------------------------------------------------------------------------------------------------------|
| Language      | Enma, tracking the newest C++ standard semantics (no pinned edition); pin the SDK release in the host repo |
| Platform      | Windows x64 (the compiler emits x64 machine code)                                                          |
| Host compiler | MSVC <code>cl</code> or <code>clang-cl</code> , C++20 or newer (needed for <code>ENMA_SIG</code> )         |
| SDK libraries | MSVC-ABI static libraries; one matching the host's CRT ( <code>/MT</code> or <code>/MD</code> )            |
| Host build    | Premake5, per the <a href="#">C++ guide</a>                                                                |
| Scripts       | <code>.em</code> source; <code>.emb</code> precompiled artifacts for shipping                              |
| Editor        | Enma Studio or an editor wired to the Enma language server (diagnostics, completion, navigation)           |

### Rules

- **MUST** ENMA-02.1 link the SDK library that matches the host's CRT setting. Mixing `/MT` and `/MD` fails at link time; pick one for the whole host.
- **MUST** ENMA-02.2 give every thread that calls `compile()` (or `compile_file`, `link`, `reload`) at least **2 MB** of stack. Raise the main thread's reserve with the linker's `/STACK` option and create compile threads with an explicit stack size.
- **MUST** ENMA-02.3 compile only the addon source files the host registers; unused addons stay out of the build.
- **MUST** ENMA-02.4 keep the SDK version pinned (vendored or by exact release) and upgrade it deliberately, reading the changelog, since the language tracks the newest C++ semantics.
- **SHOULD** ENMA-02.5 turn on the language server in every editor used for `.em` files, and treat its diagnostics as build errors.
- **SHOULD** ENMA-02.6 keep `set_optimize(e, true)` (the default) in every build; use `set_debug(e, true)` only in development hosts.

# 03

## Project layout

---

```
project/
  premake5.lua
  Project/                # Host application (C++)
    src/Project/
      Scripting/
        ScriptEngine.cpp    # Owns the engine: config, addons, permissions
        ScriptEngine.test.cpp # Tests beside the code
        Natives/            # One file per native area: InputNatives.cpp
        Addons/             # Custom addons
  scripts/                # Enma sources
    Lib/                  # Shared modules imported by others
      MathUtils.em
      MathUtils.test.em    # Script tests beside the script
    Gameplay/
      Spawner.em
      Main.em              # Entry module: int64 main()
  build/emb/              # Generated .emb artifacts (not committed)
  vendor/enma/            # Pinned SDK: headers, static libs, addon sources
```

### Rules

- **MUST** ENMA-03.1 keep Enma sources under one `scripts/` root and register it with `add_module_path`, so imports resolve the same way in every build.
- **MUST** ENMA-03.2 keep tests beside the code they test: `<name>.test.em` for scripts and `<File>.test.cpp` for host code.
- **MUST NOT** ENMA-03.3 commit `.emb` artifacts; build them in CI from `.em` sources and ship them as release assets.
- **MUST** ENMA-03.4 keep every native registration in the host's `Scripting/Natives/` folder, one file per area, so the full script-visible surface can be reviewed in one place.
- **SHOULD** ENMA-03.5 keep one module per file and name the file after the namespace it provides (`MathUtils.em` → `MathUtils::`).
- **SHOULD** ENMA-03.6 keep `Main.em` (or the host's named entry functions) thin: it wires modules together and holds no game or business logic.

# 04

## Naming

Scripts use the [C++ guide's naming](#), so host and script code read the same. The Enma standard surface (`println`, `to_string`, `push_back`) stays snake\_case because it is Enma's, not ours; the contrast shows at a glance which calls are your own.

| THING                         | CONVENTION                                                    | EXAMPLE                                                     |
|-------------------------------|---------------------------------------------------------------|-------------------------------------------------------------|
| Module file                   | PascalCase <code>.em</code> , one module per file             | <code>MathUtils.em</code> , <code>MathUtils.test.em</code>  |
| Module namespace              | PascalCase, comes from the file name                          | <code>MathUtils::Clamp()</code>                             |
| Struct / class / enum / alias | PascalCase noun                                               | <code>SpawnPoint</code> , <code>enum class WaveState</code> |
| Enumerator                    | PascalCase                                                    | <code>WaveState::Running</code>                             |
| Function / method             | PascalCase verb phrase                                        | <code>SpawnWave()</code> , <code>GetHealth()</code>         |
| Boolean query                 | <code>Is</code> / <code>Has</code> / <code>Can</code>         | <code>IsAlive()</code> , <code>HasTarget()</code>           |
| Local / parameter             | camelCase                                                     | <code>waveSize</code> , <code>outPoints</code>              |
| Data member                   | <code>m_</code> + PascalCase                                  | <code>m_Health</code> , <code>m_NextId</code>               |
| Plain-data struct field       | PascalCase, no <code>m_</code>                                | <code>struct Hit { int64 Damage; };</code>                  |
| Static member / global        | <code>s_</code> / <code>g_</code> + PascalCase (and rare, §9) | <code>s_Instance</code> , <code>g_Score</code>              |
| Compile-time constant         | PascalCase <code>constexpr</code>                             | <code>constexpr int64 MaxWaves = 10;</code>                 |
| Template parameter            | <code>T</code> or <code>T</code> + PascalCase                 | <code>T</code> , <code>TItem</code>                         |
| Macro                         | Host prefix + UPPER_SNAKE                                     | <code>PRJ_DEBUG</code>                                      |
| Host-registered native        | PascalCase, area first                                        | <code>InputIsDown</code> , <code>AudioPlay</code>           |
| Custom addon                  | PascalCase area                                               | <code>Net</code> , <code>Telemetry</code>                   |
| Test function                 | <code>Test</code> + behaviour                                 | <code>TestWaveSizeGrows()</code>                            |

## Rules

- **MUST NOT** ENMA-04.1 name anything with a leading double underscore; `__` is reserved for Enma intrinsics (`__asm_rdtsc`, `__reflect_count`).
- **MUST NOT** ENMA-04.2 shadow standard-surface names or builtins (`println`, `assert`, `time_ms`, `set_budget`).
- **MUST** ENMA-04.3 start every native's name with its area (`Input`, `Audio`, `World`) and register each area from one host file, so a reader knows the call crosses into the host.
- **MUST** ENMA-04.4 keep the module file name and its namespace identical; an `import` alias (`import "Lib/MathUtils.em" as Mu;`) is only for resolving a clash.

# 05

## Formatting and linting

---

- **MUST** ENMA-05.1 format `.em` files with `clang-format` using the same `.clang-format` as the host (2-space indent, 100 columns, attached braces); Enma's syntax is close enough to C++ that the formatter handles it. Exclude files that use `match` expressions if the formatter mangles them, and format those by hand to the same style.
- **MUST** ENMA-05.2 treat every compiler and language-server diagnostic as an error. A script that compiles with warnings does not merge.
- **MUST** ENMA-05.3 run a compile of every `.em` file in CI, through the same host configuration (addons, permissions, defines) that ships.
- **MUST** ENMA-05.4 run the review checklist from Enma's own guidance on every change (Appendix C): no `short` / `long` / `unsigned`, no standard-header `#include`, no `→` on values, only `std::` names the STD Library page lists, every `new` paired with a `delete`.
- **SHOULD** ENMA-05.5 brace every `if`, `for` and `while` body, even one-liners.
- **SHOULD** ENMA-05.6 keep one declaration per line; Enma allows `int64 x = 1, y, z = 3;` but it hides the uninitialised `y`.
- **SHOULD** ENMA-05.7 order a module: imports, `#include`s, `using` aliases, constants, types, then functions, with `main` last in an entry module.

# 06

## Types and values

---

### Sized types

- MUST** ENMA-06.1 use Enma's sized types: `int8` – `int64`, `uint8` – `uint64`, `float32`, `float64`, `bool`, `char`, `wchar`. **MUST NOT** write `short`, `long`, `unsigned` or `signed`; they are not type names.
- SHOULD** ENMA-06.2 use `int64` for counts, sizes, IDs and anything returned to the host; use narrower types only for storage that is packed or sent somewhere.
- AVOID** ENMA-06.3 the aliases `int`, `float` and `double` in new code. They are legal (`int32`, `float32`, `float64`), but writing the width removes any doubt about what size a value is. Use the `<cstdint>` spellings (`int32_t`, `size_t`) only in code shared with C++.
- MUST** ENMA-06.4 remember unsuffixed literals: integers are `int32`, `1.5` is `float64`, `1.5f` is `float32`. Suffix wide constants (`18446744073709551615ull`).

### Conversions

Enma converts more freely than C++: signed/unsigned conversions wrap, `float` to `int` truncates, and `int64` to `int32` wraps at 32 bits, all implicitly.

- MUST** ENMA-06.5 make every narrowing or sign-changing conversion explicit with `static_cast`, even where Enma would accept it silently. The cast is the reviewer's signal that the wrap is intended.
- MUST** ENMA-06.6 use brace initialisation for new variables (`int32 x{42};`). Braces reject narrowing (`uint8 u{-1};` is a compile error), which catches the silent wraps above.
- MUST** ENMA-06.7 use `enum class` for new enums; convert to integers only with `static_cast`.

### Strings, optionals and atomics

- MUST** ENMA-06.8 use `std::string` / `std::wstring` for owned text; there is no bare `string`. `const char*` is only for literals and native signatures.
- MUST** ENMA-06.9 build output text before printing: `println` takes one `std::string` or `const char*`, so use `std::format` or `std::to_string`.
- SHOULD** ENMA-06.10 use raw strings for paths and patterns: `R"(C:\dir\file)"`.
- SHOULD** ENMA-06.11 return `std::optional<T>` for "maybe a value" instead of sentinel values like `-1`.
- MUST** ENMA-06.12 use `aint8` – `aint64` for integer globals touched by more than one context (§15); compound operations on them are atomic.

```

using EntityId = int64;

enum class WaveState { Idle, Running, Done };

std::optional<EntityId> FindTarget(const std::vector<EntityId>& ids, EntityId wanted) {
    for (const EntityId& id : ids) {
        if (id == wanted) {
            return id;
        }
    }
    return {};
}

int64 main() {
    int32 waves{3};
    int64 total = static_cast<int64>(waves) * 1000;
    println(std::format("total {}", total));
    return 0;
}

```

# 07

## Functions, lambdas and control flow

---

### Functions

- **MUST** ENMA-07.1 declare the entry point as `int64 main()` (or `int32 main()`): no parameters, not `static`.
- **SHOULD** ENMA-07.2 keep functions short (about 40 lines) with at most four parameters; group more into a struct.
- **MUST** ENMA-07.3 pass large structs and containers by `const&`; Enma structs are value types and copy on pass-by-value.
- **MUST** ENMA-07.4 keep deduced (`auto`) return types consistent across every `return`. When a function mixes widths, declare the return type instead of relying on deduction.
- **MUST NOT** ENMA-07.5 declare overloads that differ only in return type.
- **MUST NOT** ENMA-07.6 rely on argument evaluation order even though Enma fixes it left to right (unlike C++); a call like `Pair(Next(), Next())` reads as a bug to anyone who knows C++. Evaluate into named locals first.
- **SHOULD** ENMA-07.7 use default parameters only for genuinely optional settings, and never refer to a later parameter.

### Lambdas

- **MUST** ENMA-07.8 use `auto` or `delegate` to hold capturing lambdas; only non-capturing lambdas convert to function pointers (`+[] { ... }`).
- **MUST NOT** ENMA-07.9 return a lambda that captures a local by reference; the compiler rejects it, and the by-value case with a pointer to a local is not caught. Capture values, or own the data with `new`.
- **SHOULD** ENMA-07.10 keep lambdas under about ten lines; name anything longer.

### Control flow

- **SHOULD** ENMA-07.11 use `match` for value-to-value mappings, always with a `_` arm:

```
int32 PointsFor(int32 tier) {
    return match (tier) {
        1 => 10,
        2 => 25,
        3 => 50,
        _ => 0
    };
}
```

- **SHOULD** ENMA-07.12 use range `for` with `const&` ( `for (const auto& [name, score] : scores)` ) instead of index loops when the index is not needed.
- **SHOULD** ENMA-07.13 use `defer { ... }` right after acquiring a resource that has no destructor of its own; it runs at scope exit and during unwinding.
- **MUST NOT** ENMA-07.14 use `goto` except to break out of nested loops, and never into a `try`, handler or `defer` body (the compiler refuses it).
- **MUST** ENMA-07.15 return early instead of nesting; no `else` after `return`.

# 08

## Structs, classes and templates

### Structs and classes

- **MUST** ENMA-08.1 use `struct` for plain data with public PascalCase fields and no invariants, and `class` for types with private state (`m_` members) and methods that keep invariants. They differ only in default access, so the keyword is how you signal intent.
- **MUST** ENMA-08.2 declare objects on the stack as `T x;` or `T x{...};`, and on the heap as `T* x = new T(...);`. `T x = new T();` is rejected; `T x();` declares a function.
- **MUST** ENMA-08.3 use `.` on values and references and `→` only on pointers or types that declare `operator→()`; the wrong one is a compile error.
- **MUST** ENMA-08.4 give every member a default initialiser or set it in the constructor's initialiser list; aggregates with excess initialisers are rejected.
- **SHOULD** ENMA-08.5 keep inheritance single and shallow. Use `virtual` only when you need runtime dispatch, mark overrides with `override`, and give polymorphic bases a `virtual` destructor.
- **AVOID** ENMA-08.6 multiple and virtual inheritance; Enma supports them, but a non-virtual diamond silently creates two base subobjects.
- **MUST** ENMA-08.7 convert class types at the call site; Enma does not apply class-to-class user conversions to function arguments.
- **MUST NOT** ENMA-08.8 make const data members in classes that need copy assignment; Enma removes copy assignment from such classes.

```
class Spawner {
public:
    explicit Spawner(int64 maxAlive) : m_MaxAlive(maxAlive) {}

    bool CanSpawn() const { return m_Alive < m_MaxAlive; }

    void OnSpawned() { m_Alive += 1; }

private:
    int64 m_MaxAlive = 0;
    int64 m_Alive = 0;
};
```

## Templates

- **SHOULD** ENMA-08.9 use templates for genuinely generic code (containers, math over numeric types) and constrain them with concepts.
- **MUST** ENMA-08.10 write `typename` before a dependent type name inside a template ( `typename T::Type` ).
- **MUST** ENMA-08.11 remember that two instantiations of one template are different types: a value crossing a parameter, return or `new` boundary must match exactly.
- **AVOID** ENMA-08.12 deep template metaprogramming; Enma caps instantiations at 16,384 per compile and nesting at 1,024, and every instantiation is monomorphised into machine code.
- **SHOULD** ENMA-08.13 prefer `constexpr` functions and `if constexpr` over recursive template tricks for compile-time work.

# 09

## Memory and ownership

---

Enma has no garbage collector. Stack objects, `std` containers and class members are destroyed deterministically at scope exit, including during exception unwinding.

- **MUST** ENMA-09.1 prefer stack values and `std` containers. Reach for `new` only when an object must outlive its scope or be shared by address.
- **MUST** ENMA-09.2 pair every `new` with exactly one `delete` and every `new T[N]` with `delete[]`, and make the owner obvious: one struct or function owns each heap object and deletes it in its destructor or with `defer`.
- **SHOULD** ENMA-09.3 put `defer { delete p; }` on the line after a `new` whose object never leaves the function.
- **MUST NOT** ENMA-09.4 return, store in a global or field, or capture by reference the address of a local. The compiler rejects the obvious cases; its escape analysis is conservative, so do not lean on it for the rest.
- **SHOULD** ENMA-09.5 mark functions that must never leak allocations with `[[noescape]]`; the compiler then rejects any allocation that outlives the call.
- **MUST** ENMA-09.6 index with `.at(i)` when the index comes from input or computation that could be wrong. `v[i]`, `a[i]` and `p[i]` are **not** bounds-checked; a near miss is undefined behaviour.
- **MUST** ENMA-09.7 treat a pointer from a native as borrowed unless the native is documented as transferring ownership, and never `delete` it.
- **AVOID** ENMA-09.8 mutable globals. They are shared by every context of a module (§15); keep state in structs passed by reference, or in context userdata set by the host.
- **SHOULD** ENMA-09.9 check `heap_count()` in tests before and after a scenario to catch leaks.

```
int64 SumWave(const std::vector<int64>& damage) {
    int64* scratch = new int64[damage.size()];
    defer { delete[] scratch; };

    int64 total = 0;
    for (int64 i = 0; i < damage.size(); i++) {
        scratch[i] = damage.at(i) * 2;
        total += scratch[i];
    }
    return total;
}
```

# 10

## Errors and exceptions

---

### In scripts

- **MUST** ENMA-10.1 throw exceptions for failures the caller cannot reasonably ignore, and return `std::optional<T>` (or a `bool` plus out-parameter) for expected misses such as "not found".
- **MUST** ENMA-10.2 throw types derived from `std::runtime_error` with a message that names what failed and the value involved.
- **MUST** ENMA-10.3 catch by reference ( `catch (const std::runtime_error& e)` ), most specific first, with `catch (...)` last; Enma rejects misordered handlers.
- **MUST** ENMA-10.4 catch only where you can recover or add context; otherwise let the exception reach the host, where `execute()` reports it.
- **MUST NOT** ENMA-10.5 use exceptions for control flow in per-frame code.
- **MUST** ENMA-10.6 treat null dereference and division by zero as bugs, even though they are catchable checked faults. Check first; do not write code that catches them on purpose.
- **MUST** ENMA-10.7 remember that stack overflow, wild out-of-bounds writes and use-after-free that hits unmapped memory are **not** catchable: they end the run without unwinding.

### In hosts

- **MUST** ENMA-10.8 check the result of every SDK call: `compile()` and `deserialize()` return `nullptr`, `execute()` and `call()` return `false` on failure.
- **MUST** ENMA-10.9 report compile failures with `last_error()` — code, message, file, line and column — so the script author sees where the problem is.
- **MUST** ENMA-10.10 keep the host running when a script fails: log the error, mark the script as failed, and carry on.
- **MUST** ENMA-10.11 report faults from natives with `runtime_error(message)` (or `runtime_exception`) rather than C++ exceptions, and keep every native `noexcept` at its boundary.
- **MUST** ENMA-10.12 call `mark_native_raises(e, name, false)` only for natives that genuinely can never raise; the per-call exception check is cheap insurance.

```
enma::module_t* mod = enma::compile(m_Engine, source.data(), source.size(), "Spawner.em");
if (mod == nullptr) {
    const auto& err = enma::last_error(m_Engine);
    PRJ_LOG_ERROR("{}: {}: {}: {}", err.file, err.line, err.column, err.message);
    return std::unexpected(ScriptError::CompileFailed);
}
```

# Modules, namespaces and the preprocessor

## Imports

- **MUST** ENMA-11.1 share code between files with `import`, not `#include`. Imported declarations keep their exact types and are reached through the module's namespace (`MathUtils::Clamp`).
- **MUST** ENMA-11.2 import by module name (`import MathUtils`) when the module is on a host-registered module path, and by quoted relative path (`import "Lib/MathUtils.em";`) only for files that are not.
- **SHOULD** ENMA-11.3 alias an import (`as Mu`) only to resolve a name clash or to shorten a deep path; never alias the same file twice.
- **MUST NOT** ENMA-11.4 `#include` standard headers; the standard surface is always available. `#include` is only for sharing macros between your own files.
- **MUST** ENMA-11.5 keep a module's private helpers `static` or in an anonymous namespace so they do not leak to importers.

## Namespaces

- **MUST** ENMA-11.6 keep each module's public declarations inside the namespace its file name gives it; do not add extra top-level namespaces in a module.
- **SHOULD** ENMA-11.7 use `using namespace` only inside function bodies, and namespace aliases (`namespace Rd = Engine::Render::Detail;`) for long paths. A namespace alias cannot point at an imported module.

## Preprocessor

- **AVOID** ENMA-11.8 macros in scripts. Use `constexpr` constants, `if constexpr` and functions instead.
- **MAY** ENMA-11.9 use `#ifdef` on host-supplied defines (`define(e, "PRJ_DEBUG", "1")`) to compile debug-only checks.
- **MUST** ENMA-11.10 prefix any macro with the host prefix (`PRJ_`), and keep macros in one shared include file.

## Precompiled modules

- **MUST** ENMA-11.11 ship scripts as `.emb` artifacts built in CI with `serialize()`, and load them with `deserialize()`, checking for `nullptr`.
- **SHOULD** ENMA-11.12 build `.emb` files with `keep_debug = true` for development and internal builds, and `false` only for public releases where the source must not be recoverable.

- **MUST** ENMA-11.13 rebuild every `.emb` when the SDK version changes.

# 12

## Annotations and data layout

Enma accepts a closed set of annotations, each only in its own position; anything else is a compile error.

| ANNOTATION                                             | ON                              | USE IT WHEN                                                                 |
|--------------------------------------------------------|---------------------------------|-----------------------------------------------------------------------------|
| <code>[[packed]]</code>                                | struct                          | Matching a wire or file format byte for byte                                |
| <code>[[align(N)]]</code>                              | struct, member                  | Data handed to SIMD code or hardware that needs alignment                   |
| <code>[[offset(N)]]</code>                             | member                          | Mirroring a native struct layout exactly                                    |
| <code>[[reflect]]</code>                               | struct                          | Tools, editors or serializers that enumerate fields                         |
| <code>[[serialize]]</code>                             | struct                          | Generating <code>to_bytes</code> / <code>from_bytes</code> for flat records |
| <code>[[no_vtable]]</code>                             | struct                          | Rare; dropping the dispatch table from a type with virtuals                 |
| <code>[[inline]]</code> /<br><code>[[noinline]]</code> | function                        | A profile shows the call matters                                            |
| <code>[[noopt]]</code>                                 | function                        | Debugging only; never committed                                             |
| <code>[[noescape]]</code>                              | function                        | Guaranteeing a function leaks no allocation                                 |
| <code>[[dll(...)]]</code>                              | <code>extern</code><br>function | Binding a DLL export; needs <code>PERM_FFI</code>                           |

- **MUST** ENMA-12.1 use layout annotations (`[[packed]]`, `[[align]]`, `[[offset]]`) only for types that cross a boundary (files, network, natives), with a comment naming the format they match.
- **MUST NOT** ENMA-12.2 commit `[[noopt]]`.
- **SHOULD** ENMA-12.3 add `[[reflect]]` to structs that host tools or editors inspect, instead of hand-written field tables.
- **MUST** ENMA-12.4 keep `[[serialize]]` structs flat: no bit-fields, inline arrays or nested structs, and no pointers (their addresses would be written out).
- **MUST NOT** ENMA-12.5 use `[[dll(...)]]` in gameplay or user scripts. FFI belongs in host natives, where it can be reviewed; scripts that need it are compiled only by engines granted `PERM_FFI`.

- **MUST** ENMA-12.6 remember that the standard attributes (`[[nodiscard]]`, `[[likely]]`, `[[deprecated]]` and others) are parsed and **ignored**; do not rely on them to enforce anything.

```
// Matches the 6-byte packet header sent by the host's net layer.  
[[packed]] struct PacketHeader {  
    uint8 Type;  
    uint8 Flags;  
    int32 Length;  
};
```

# 13

## Embedding: engine, module and context lifecycle

| OBJECT                                     | CREATED BY                                                                                         | OWNS                                                                       | LIFETIME                            |
|--------------------------------------------|----------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|-------------------------------------|
| <b>Engine</b><br>( <code>engine_t</code> ) | <code>enma::create()</code>                                                                        | Configuration, registered natives and addons, permissions, compile state   | The host's scripting subsystem      |
| <b>Module</b><br>( <code>module_t</code> ) | <code>compile</code> , <code>compile_file</code> ,<br><code>deserialize</code> , <code>link</code> | Compiled code, script globals, instruction budget, debug hook, breakpoints | Until reloaded or destroyed         |
| <b>Context</b>                             | <code>create_context(mod)</code>                                                                   | Heap, stack, RNG, in-flight exception, 16 userdata slots                   | One per thread that runs the module |

### Rules

- **MUST** ENMA-13.1 wrap the engine, each module and each context in an RAIi owner in the host (`ScriptEngine`, `ScriptModule`, `ScriptContext`) that releases in reverse order: contexts, then modules, then the engine.
- **MUST** ENMA-13.2 configure the engine completely — optimisation, debug, defines, include and module paths, resolvers, permissions, addons, natives — **before** the first compile. Configuration after compiling is a bug.
- **MUST** ENMA-13.3 destroy every module and context before `destroy(engine)`; the engine does not track its children.
- **MUST** ENMA-13.4 call `enma::shutdown()` once when the scripting subsystem is torn down; in a DLL that may unload, call it from `DLL_PROCESS_DETACH` and run no script code afterwards.
- **SHOULD** ENMA-13.5 load scripts through `set_import_resolver` / `set_include_resolver` backed by the host's virtual file system, so packaged builds and development builds resolve the same way.
- **MUST** ENMA-13.6 run entry points by name with `execute()` or `call()`, and read results with the matching `return_value`, `return_string` or `return_float`.
- **SHOULD** ENMA-13.7 use `reload` for hot reload in development builds only, and drop every cached pointer into the old module (function addresses, annotation info, strings) after a reload.

```

namespace Project {

std::expected<void, ScriptError> ScriptEngine::Init() {
    m_Engine = enma::create();
    enma::set_optimize(m_Engine, true);
    enma::set_debug(m_Engine, PRJ_DEBUG_BUILD);
    enma::set_permissions(m_Engine, 0);           // nothing extra by default
    enma::register_addon_core(m_Engine);          // only the addons scripts use
    enma::register_addon_std(m_Engine);
    enma::add_module_path(m_Engine, "scripts");
    RegisterInputNatives(m_Engine);
    RegisterAudioNatives(m_Engine);
    return {};
}

ScriptEngine::~~ScriptEngine() {
    m_Modules.clear();                           // destroys contexts, then modules
    if (m_Engine != nullptr) {
        enma::destroy(m_Engine);
    }
}

} // namespace Project

```

# 14

## Native functions and addons

---

Natives are the script-visible API of the host. Review them like a public API: each one is a door out of the sandbox.

### Registering natives

- **MUST** ENMA-14.1 register natives with `enma::register_typed<&Fn>(e, signature)` where possible, and wrap every signature in `ENMA_SIG("...")` so a malformed signature fails the host build instead of logging at run time.
- **MUST** ENMA-14.2 use Enma's sized types in signatures ( `int64` , `float64` , `const char*` ) and keep them to plain values, pointers and references; signatures with `struct` , `class` , lambdas or varargs are refused.
- **MUST** ENMA-14.3 keep every native function alive for as long as any module that calls it exists; the engine does not own `fn_ptr` .
- **MUST** ENMA-14.4 validate every argument inside the native: natives are not protected by Enma's checks. Use `heap_is_tracked(ptr)` before trusting a pointer that claims to be script heap memory.
- **MUST** ENMA-14.5 declare ownership: `mark_native_returns_borrow` for handles the host keeps, `mark_native_deallocates` for natives that free a parameter. Undeclared ownership is a leak or a double free waiting to happen.
- **MUST** ENMA-14.6 gate natives that touch files, the network, processes or other unsafe areas with a permission flag at registration ( `PERM_FILE` , `PERM_FFI` , or a host-defined bit), so scripts compiled without the grant fail to compile.
- **SHOULD** ENMA-14.7 keep natives coarse: one call that does a whole job ( `WorldSpawnWave(count)` ) beats ten fine-grained calls per frame.

```

namespace Project::Natives {

static bool InputIsDown(int64_t key) noexcept {
    if (key < 0 || key ≥ Input::KeyCount) {
        enma::runtime_error("InputIsDown: key out of range");
        return false;
    }
    return Input::IsDown(static_cast<Key>(key));
}

void RegisterInputNatives(enma::engine_t* e) {
    enma::register_typed<&InputIsDown>(e, ENMA_SIG("bool InputIsDown(int64 key)"));
}

} // namespace Project::Natives

```

## Addons

- **MUST** ENMA-14.8 register only the addons a product needs (`register_addon_core`, `register_addon_std`, ...) instead of `register_all_addons` in shipping hosts, and disable the filesystem addon unless scripts truly need it.
- **MUST** ENMA-14.9 build host-specific capabilities (networking, telemetry, game systems) as custom addons with their own namespace, not as loose natives, when they have more than a handful of functions.
- **MUST** ENMA-14.10 document every addon's functions, permission and threading rules in its header and on the docs site.
- **MUST NOT** ENMA-14.11 expose anything from an addon that lets a script register further natives or change permissions.

# 15

## Threading and contexts

---

- **MUST** ENMA-15.1 compile on one thread per engine at a time. Concurrent `compile`, `link`, `reload` or `deserialize` calls on one engine are refused; use one engine per compiling thread or serialise compiles.
- **MUST** ENMA-15.2 give every thread that runs a module its own context from `create_context(mod)`, created before the thread starts and destroyed after it is joined.
- **MUST NOT** ENMA-15.3 share a context between threads at the same time; a second thread entering a busy context is refused. Hand a context to another thread only after the first has finished and been joined.
- **MUST** ENMA-15.4 treat script globals as shared by every context of a module. Guard them with `aint*` types or a `mutex` + `lock_guard` from the Thread addon, or write them from one thread only.
- **MUST** ENMA-15.5 open an `execution_scope` on host threads that call into script memory outside a run (allocating on the script heap, invoking a script function pointer, raising an exception).
- **SHOULD** ENMA-15.6 give workers that need their own budget, breakpoints or globals their own module; budgets and debug state are per module, not per context.
- **MUST NOT** ENMA-15.7 use script threads (the Thread addon) for work the host can schedule; prefer host-side job systems that call into contexts.
- **MUST** ENMA-15.8 remember that `async/await`, futures and coroutines do not exist in Enma; long work is split across frames or moved to a worker context.

```
// One module, one context per worker.
std::vector<enma::context_t*> contexts;
for (int32_t i = 0; i < workerCount; ++i) {
    contexts.push_back(enma::create_context(mod));
}
// ... start workers, each calling enma::call(contexts[i], ...) ...
// ... join workers ...
for (enma::context_t* ctx : contexts) {
    enma::destroy_context(mod, ctx);
}
```

# 16

## Safety: permissions, budgets and containment

Enma's containment comes from the host: scripts reach only what is registered, permission flags gate dangerous features at compile time, and budgets cap what a script can consume.

### Permissions

| FLAG                                            | GATES                                                            | DEFAULT IN <code>PROJECT</code>                             |
|-------------------------------------------------|------------------------------------------------------------------|-------------------------------------------------------------|
| <code>PERM_FFI</code><br>( <code>0x01</code> )  | <code>[[dll(...)]]</code> externs and natives registered with it | Off everywhere; on only for trusted tooling hosts           |
| <code>PERM_FILE</code><br>( <code>0x02</code> ) | All Filesystem addon operations                                  | Off for user or untrusted scripts; on for first-party tools |
| <b>Host-defined bits</b>                        | Natives the host gates itself                                    | Documented beside each native                               |

- **MUST** ENMA-16.1 start every engine with `set_permissions(e, 0)` and grant flags explicitly, per engine, for the scripts that engine compiles.
- **MUST** ENMA-16.2 compile untrusted scripts (user content, mods, downloaded code) on a separate engine with no permissions, no filesystem addon and only the natives they are meant to have.
- **MUST** ENMA-16.3 remember that permissions check what a module binds, not what its machine code does: never load `.emb` files from untrusted sources without checking their origin, since they are native code.

### Budgets

- **MUST** ENMA-16.4 set an instruction budget ( `set_budget` ) and a heap budget ( `set_memory_budget` ) on every module that runs untrusted or user-editable scripts, and treat exceeding them as a failed run.
- **SHOULD** ENMA-16.5 set budgets from the host, not from the script; a script can call `set_budget` itself, so untrusted scripts must not be able to raise their own limits (do not rely on scripts calling it).
- **SHOULD** ENMA-16.6 size budgets from measurements of real content plus headroom, and log the measured use in development builds.

## Containment checklist

- Natives validate every argument and never trust script pointers without `heap_is_tracked`.
- No native lets a script register code, change permissions or reach the host's allocator directly.
- Every script run has a timeout path: budgets for untrusted code, a watchdog for long-running trusted code.
- A failed `execute()` disables that script and leaves the host running.

# 17

## Performance

---

Enma compiles whole modules to native x64, so script code runs at C++-like speed. Most slowness comes from crossing the host boundary or from allocation.

- **MUST** ENMA-17.1 measure before optimising: time with `time_ms()` or `__asm_rdtsc()` in scripts, and the host profiler around `execute()` / `call()`.
- **SHOULD** ENMA-17.2 compile modules once and cache them (or ship `.emb` files); `compile()` is far more expensive than running.
- **SHOULD** ENMA-17.3 create contexts once per worker and reuse them, rather than creating one per call.
- **SHOULD** ENMA-17.4 keep per-frame script code free of heap allocation: reuse vectors (`clear()` keeps capacity), pass by `const&`, and avoid building strings unless they are printed.
- **SHOULD** ENMA-17.5 batch native calls: one call that processes an array is cheaper than one call per element.
- **MAY** ENMA-17.6 use the SIMD and Math:3D addons, `[[align(16)]]` data and `[[inline]]` on hot paths, after a profile shows the need.
- **SHOULD** ENMA-17.7 use `hold_compile_caches(true)` around a batch of compiles (for example, loading a level's scripts), paired with exactly one `hold_compile_caches(false)`.
- **AVOID** ENMA-17.8 deep template instantiation in hot modules; every instantiation is separate machine code, which grows the module and compile time.

# 18

## Perception: scope, script structure and lifecycle

Perception is the host these scripts run in: it loads Enma scripts, gives them a process API, a render API and the default Enma addons, and runs their routines every frame. Sections 18–21 apply to Perception scripts; everything before them still applies.

### Scope

- MUST** ENMA-18.1 use Perception scripts only with single-player, offline games and your own software. **MUST NOT** attach to games with online play, shared leaderboards or anti-cheat, or to any process you do not have the right to modify.
- MUST** ENMA-18.2 check the target at startup (process name and module) and unload if it is not the expected single-player title.

### Lifecycle

| STAGE       | FUNCTION                                         | RULES                                                                                                                                  |
|-------------|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Load        | <code>int64 main()</code>                        | Attach to the process, create fonts and resources, register routines. Return <code>1</code> to stay loaded, <code>≤ 0</code> to unload |
| Every frame | Routines: <code>void Callback(int64 data)</code> | Read state, draw, return promptly                                                                                                      |
| Unload      | <code>void unload()</code> (optional)            | Runs after every routine has stopped; release what <code>main</code> set up and restore anything you changed                           |

- MUST** ENMA-18.3 do one-time work (attaching, pattern scans, font creation, shader and buffer creation) in `main()`, never in a routine.
- MUST** ENMA-18.4 return `0` from `main()` with a logged reason when setup fails (process not found, pattern not found), instead of leaving routines running against bad state.
- MUST** ENMA-18.5 keep each routine short and non-blocking; split slow work across frames.
- MUST** ENMA-18.6 treat routines as concurrent: they may run at the same time and share globals, so shared mutable state is atomic (`aint64`) or behind a `mutex`.
- MUST** ENMA-18.7 restore every value a script wrote to the target in `unload()`, so unloading leaves the game as it was.

- **SHOULD** ENMA-18.8 keep each routine's handle and unregister it when its feature is turned off, rather than branching on a flag every frame.

```
std::optional<proc_t> g_Game;
int64 g_HudRoutine = -1;
int64 g_Font = 0;

void DrawHud(int64 data) {
    using namespace render;
    if (!g_Game || !g_Game->alive()) {
        return;
    }
    draw_text("trainer active", vec2(40.0f, 40.0f), color(255, 255, 255, 255), g_Font, 1, 1.0f);
}

int64 main() {
    using namespace render;
    g_Game = ref_process("game.exe");
    if (!g_Game) {
        println("game.exe not running");
        return 0;
    }
    g_Font = create_font("Inter", 20);
    if (g_Font == 0) {
        println(std::format("create_font failed: {}", get_last_render_error()));
        return 0;
    }
    g_HudRoutine = register_routine(reinterpret_cast<int64>(DrawHud), 0);
    return g_HudRoutine == -1 ? 0 : 1;
}

void unload() {
    g_Game.reset();
}
```

Check each call's exact signature against the Perception API pages for your version; the shape above is the pattern to follow.

# 19

## Perception: process access and permissions

`proc_t` is a move-only reference to a process that releases itself when it leaves scope. Reads are always available; writes and heavier operations are gated by permissions.

| PERMISSION                             | GATES                                                                                | DEFAULT FOR SINGLE-PLAYER SCRIPTS                                                    |
|----------------------------------------|--------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| (none)                                 | Attaching, reading memory, modules, exports, pattern and value scans, memory regions | Allowed                                                                              |
| <code>write_memory</code>              | <code>w*</code> writes, <code>wvm</code>                                             | Only for scripts whose purpose is to change values (trainers), and only those values |
| <code>virtual_memory_operations</code> | <code>alloc_vm</code> , <code>free_vm</code> in the target                           | Off; single-player mods rarely need it                                               |
| <code>kernel_rw_access</code>          | The <code>kernel</code> namespace, physical memory, DTB overrides                    | <b>Off.</b> Single-player games do not need kernel access                            |

### Rules

- MUST** ENMA-19.1 request only the permissions a script uses, and state why at the top of the script in a comment.
- MUST NOT** ENMA-19.2 use `kernel_rw_access`, DTB overrides or physical-memory reads and writes. If a target only works through them, it has protections you should not be working around.
- MUST** ENMA-19.3 check `alive()` before using a stored `proc_t` in a routine; the game may have closed.
- MUST** ENMA-19.4 check every pointer read from the target with `is_valid_address` before following it, and treat a zero or invalid pointer as "not ready", not as an error to retry every frame.
- MUST** ENMA-19.5 locate code and data with `find_code_pattern` (byte signatures with `??` wildcards) and module-relative offsets, never with absolute addresses.
- MUST** ENMA-19.6 keep signatures and offsets in one table at the top of the script (or a shared `Offsets.em` module), each with the game version it was verified on, and fail closed in `main()` when any of them is not found.
- SHOULD** ENMA-19.7 read structs in one bulk call (`rvm`) instead of many single reads, and cache values that do not change every frame.

- **MUST** ENMA-19.8 write the smallest change possible, remember the original value, and restore it in `unload()` and when the feature is turned off.

```
// Offsets.em: verified on game.exe 1.4.2
namespace Offsets {
constexpr const char* PlayerManager = "48 8B 05 ?? ?? ?? ?? 48 85 C0 74 ?? 8B 40 10";
constexpr int64 Health = 0x10;
constexpr int64 Ammo = 0x24;
}
```

# 20

## Perception: rendering and overlays

---

The render API (namespace `render`) records draw calls during a routine and executes them at the end of the frame.

- **MUST** ENMA-20.1 create fonts, bitmaps, shaders and buffers once in `main()` and reuse them. Load each font at the size it is drawn; fonts and bitmaps are cached for the whole process and never freed.
- **MUST** ENMA-20.2 check every `create_*` / `load_*` handle: `0` means failure. Read `get_last_render_error()` and fail setup with its message.
- **MUST** ENMA-20.3 use `color(...)` for RGBA values and `vec2(...)` for positions and sizes; keep shared colours as named constants (`Colors::Text`, `Colors::Warning`).
- **SHOULD** ENMA-20.4 batch similar shapes with the batch forms (`draw_rects`, `draw_lines`) instead of one call per item.
- **SHOULD** ENMA-20.5 upload static geometry once to resident buffers and draw it with `custom_draw_buffer*`; use dynamic buffers only for data that changes every frame.
- **MUST** ENMA-20.6 set pipeline state (viewport, blend, targets) right before each custom draw, and end every custom sequence with `custom_restore_state()`; state does not carry over between draws or routines.
- **SHOULD** ENMA-20.7 keep overlays readable and out of the way: one corner by default, outlined or shadowed text (`effect` 1 or 2), and a toggle to hide everything.
- **SHOULD** ENMA-20.8 measure overlay cost with `get_render_thread_frame_ms()` and GPU timers, and keep a script's drawing well under a millisecond per frame.
- **MAY** ENMA-20.9 build settings panels with the GUI API rather than hand-drawn widgets, so every script's controls look and behave the same.

# 21

## Perception: API conventions and tooling

---

Perception's own conventions, which every script follows:

- **MUST** ENMA-21.1 use `static_cast<T>(x)` for numeric conversions and `reinterpret_cast<int64>(fn)` to pass a function as a routine or callback handle.
- **MUST** ENMA-21.2 suffix `float32` literals with `f` (`0.2f`, `40.0f`); a bare `0.2` is `float64`.
- **MUST** ENMA-21.3 treat values from `create_*` and `load_*` as opaque handles and pass them only to the matching API.
- **MUST** ENMA-21.4 use `std::string` for owned text and `const char*` for read-only parameters; convert numbers with `std::to_string` or `std::format`.
- **MUST** ENMA-21.5 use `std::vector<T>` for dynamic storage and `std::array<T, N>` for fixed-size storage.
- **MUST** ENMA-21.6 use `print_console` for chatty debug output so the main log stays readable, and `println` for lifecycle messages.

### Tooling

- **MUST** ENMA-21.7 edit scripts with the Perception language server enabled (diagnostics, hover, completion, go-to-definition) and fix every diagnostic before loading a script.
- **MAY** ENMA-21.8 use the Perception MCP interface to let an AI agent load, run and inspect scripts locally; review any agent-written script against this guide before running it against a game.
- **SHOULD** ENMA-21.9 keep each script in its own folder with a short `README.md`: the target game and version, the permissions it requests and why, and how to turn each feature off.

# 22

## Testing

---

### Script tests

- **MUST** ENMA-22.1 keep tests beside the script as `<Name>.test.em`, importing the module under test.
- **MUST** ENMA-22.2 write each test as an `int64 Test<Behaviour>()` function that returns `0` on success, using the `assert(cond, msg)` builtin for checks. The host's test runner finds and calls every `Test*` function.
- **MUST** ENMA-22.3 check for leaks: record `heap_count()` before and after each test and fail if it changed.
- **MUST** ENMA-22.4 test error paths too: that the right exception is thrown, and that `.at()` throws on a bad index.
- **SHOULD** ENMA-22.5 run tests on an engine configured like production (same addons, permissions and defines) plus a tighter budget, so runaway loops fail fast.

```
import "MathUtils.em";

int64 TestClampKeepsValueInRange() {
    int64 before = heap_count();
    assert(MathUtils::Clamp(15, 0, 10) == 10, "clamps above max");
    assert(MathUtils::Clamp(-3, 0, 10) == 0, "clamps below min");
    assert(MathUtils::Clamp(4, 0, 10) == 4, "keeps values in range");
    assert(heap_count() == before, "no leaks");
    return 0;
}
```

### Host tests

- **MUST** ENMA-22.6 test every native with good, boundary and bad arguments, from a script, and check that bad arguments raise instead of crashing.
- **MUST** ENMA-22.7 test that a script without a permission fails to compile when it calls a gated native.
- **MUST** ENMA-22.8 test budgets: an infinite loop and a runaway allocation each fail `execute()` while the host keeps running.
- **SHOULD** ENMA-22.9 run host and script tests under the same CI job on Windows x64, with ASan builds of the host where the SDK allows it.
- **SHOULD** ENMA-22.10 compile every `.em` file and round-trip it through `serialize` / `deserialize` in CI to catch artifact problems before release.

# 23

## Git, releases and review

---

- **MUST** ENMA-23.1 follow **Conventional Commits**: `type(scope): summary`, imperative, lower case, no trailing period, under 72 characters. Use the module or host area as scope: `feat(spawner): add wave cooldown`, `fix(natives): validate key range`.
- **MUST** ENMA-23.2 mark breaking changes to natives, addons or module APIs with `!` and a **BREAKING CHANGE** footer.
- **MUST** ENMA-23.3 keep commits small and focused, each compiling every script and passing tests.
- **MUST** ENMA-23.4 record the SDK version in the host's release notes, and rebuild every `.emb` artifact for each release.

### Review checklist: scripts

- Sized types only; no `short`, `long`, `unsigned`, or standard-header `#include`.
- Narrowing and sign changes are explicit casts; new variables use braces.
- `.` on values, `→` only on pointers; only `std::` names the STD Library page lists.
- Every `new` has an owner and a `delete`; no address of a local escapes.
- Untrusted indexes use `.at()`; globals touched by several contexts are atomic or locked.

### Review checklist: hosts

- Every native validates its arguments, declares ownership, and is gated if it touches files, network or processes.
- Signatures use `ENMA_SIG`; natives are `noexcept` and report faults with `runtime_error`.
- Engines are fully configured before compiling; contexts and modules are destroyed before engines.
- Untrusted scripts run on a separate, permissionless engine with budgets set by the host.

# A

## Appendix

---

### A. Minimal runnable script

```
// Main.em
int32 Fib(int32 n) {
    if (n ≤ 1) {
        return n;
    }
    return Fib(n - 1) + Fib(n - 2);
}

int64 main() {
    println(std::to_string(Fib(30)));
    return 0;
}
```

## B. Host skeleton

```
int main() {
    enma::engine_t* engine = enma::create();
    enma::set_permissions(engine, 0);
    enma::register_addon_core(engine);
    enma::register_addon_std(engine);

    const std::string source = Project::ReadFile("scripts/Main.em");
    enma::module_t* mod = enma::compile(engine, source.data(), source.size(), "Main.em");
    if (mod == nullptr) {
        std::println(stderr, "{}", enma::last_error_message(engine));
        enma::destroy(engine);
        return 1;
    }

    enma::context_t* ctx = enma::create_context(mod);
    const bool ok = enma::execute(ctx, "main");
    const int64_t result = ok ? enma::return_value(ctx) : -1;

    enma::destroy_context(mod, ctx);
    enma::module_destroy(mod);
    enma::destroy(engine);
    enma::shutdown();

    return ok ? static_cast<int>(result) : 1;
}
```

Check exact SDK signatures against the API Reference for your pinned SDK version.

## C. Script review checklist (from Enma's own guidance)

- ☐ No `short`, `long` or `unsigned`.
- ☐ No `#include` of a standard header.
- ☐ No `→` on a value whose type declares no `operator→()`.
- ☐ No `std::` names that the STD Library page does not list.
- ☐ Every `new` has a `delete`; nothing returns the address of a local.

## D. Sources

- [Enma documentation](#)
- [Enma full docs for assistants \(llms-full.txt\)](#)
- [Writing Enma with an Assistant](#)
- [Semantics & Limits](#)
- [SDK: Native Functions](#)

- [SDK: Threading & Concurrency](#)
- [SDK: Engine Lifecycle](#)