

# C++ Style Guide

Conventions for writing, structuring, formatting and shipping modern C++23 that builds clean on MSVC, Clang and GCC.

```
namespace Project {  
    // Clear beats clever. Measure, then optimize.  
    constexpr uint32_t MaxFramesInFlight = 3;  
} // namespace Project
```

# Inside this guide

|    |                                         |    |
|----|-----------------------------------------|----|
| 01 | Purpose and scope                       | 2  |
| 02 | Language baseline and toolchain         | 3  |
| 03 | Repository and project layout           | 4  |
| 04 | Project short code and the base header  | 6  |
| 05 | Naming                                  | 8  |
| 06 | Formatting                              | 11 |
| 07 | Headers and includes                    | 16 |
| 08 | Types, ownership and lifetimes          | 18 |
| 09 | Classes and interfaces                  | 20 |
| 10 | Functions                               | 22 |
| 11 | Error handling                          | 25 |
| 12 | const and compile-time code             | 27 |
| 13 | Templates and concepts                  | 29 |
| 14 | Memory and performance                  | 31 |
| 15 | Concurrency                             | 33 |
| 16 | Platform abstraction and API boundaries | 34 |
| 17 | Logging, assertions and diagnostics     | 36 |
| 18 | Testing                                 | 38 |
| 19 | Build system                            | 40 |
| 20 | Comments and documentation              | 42 |
| 21 | Git and review                          | 44 |
| A  | Appendix                                | 46 |

Rule levels

|      |          |        |            |     |       |
|------|----------|--------|------------|-----|-------|
| MUST | MUST NOT | SHOULD | SHOULD NOT | MAY | AVOID |
|------|----------|--------|------------|-----|-------|

# 01

## Purpose and scope

---

Every C++ project that adopts this guide is written in C++23, laid out the same way every time, and clean under MSVC even when another compiler is used day to day. The goal is that any two projects read as if one person wrote them.

The guide covers language level, layout, naming, formatting, ownership, error handling, performance, testing, builds and commits. It does not cover per-project architecture; each project documents that in its own `docs/` folder.

**Rule levels.** Each rule carries one of three words:

| LEVEL                          | MEANING                                                    | BREAKING IT                                        |
|--------------------------------|------------------------------------------------------------|----------------------------------------------------|
| <b>MUST</b><br><b>MUST NOT</b> | Non-negotiable. Tooling or review enforces it.             | Needs a written reason in the code and the commit. |
| <b>SHOULD</b>                  | The default. Expected in nearly all code.                  | Fine with a short comment saying why.              |
| <b>MAY</b>                     | Allowed when it helps; neither encouraged nor discouraged. | Nothing to break.                                  |
| <b>AVOID</b>                   | Allowed but a smell.                                       | Fine when the alternative is clearly worse.        |

**Examples.** Code samples use `PRJ` as the project short code (`prj` in lowercase contexts) and `Project` as the root namespace. Both are placeholders; substitute your own project's code and name.

**Google C++ Style Guide.** The rules on short functions, choosing names, local variables, structs vs. classes, comments, line length, non-ASCII characters, lambdas, floating-point literals, looping and branching, pointer and reference expressions, return values, preprocessor directives, class format and constructor initializer lists are adapted from the [Google C++ Style Guide](#) (CC BY 3.0). Its examples are rewritten here in this guide's naming.

**When the guide is silent,** match the surrounding code, then the Google C++ Style Guide, then the C++ Core Guidelines, in that order.

# 02

## Language baseline and toolchain

The language level is C++23, and MSVC is the compiler every line must pass. Clang and GCC are welcome as second and third opinions, but code that only builds on them is broken.

- **MUST** build with MSVC (latest Visual Studio, `/std:c++latest` until `/std:c++23` is stable) through the Premake-generated `.sln`.
- **MUST** build warning-free at `/W4` with `/permissive-`, `/Zc:__cplusplus` and `/Zc:preprocessor`.
- **SHOULD** also build with Clang (`-std=c++23 -Wall -Wextra -Wpedantic`) on macOS ARM and Linux. Treat a Clang-only warning as a real bug until proven otherwise.
- **MUST NOT** rely on compiler extensions without a wrapper macro in the base header (see section 4).
- **SHOULD** check a feature against the MSVC column of cppreference's compiler support table before using it. If MSVC lacks it, wait or wrap it.

### Standard library features in use

| USE FREELY                                                                                                                                         | USE WITH CARE                                                                             | AVOID FOR NOW                                                    |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| <code>std::expected</code> , <code>std::span</code> ,<br><code>std::string_view</code> , <code>std::optional</code> ,<br><code>std::variant</code> | <code>std::format</code> and<br><code>std::print</code> (compile time<br>cost in headers) | C++20 modules (tooling and<br>Premake support still uneven)      |
| Concepts and <code>requires</code> clauses                                                                                                         | <code>std::ranges</code> pipelines<br>(debug-build speed)                                 | <code>std::generator</code> until all<br>three compilers ship it |
| <code>constexpr</code> / <code>constexpr</code> , <code>if</code><br><code>constexpr</code>                                                        | Coroutines (only behind a<br>project-level abstraction)                                   | <code>&lt;regex&gt;</code> (slow; use a<br>vendored library)     |
| <code>std::to_underlying</code> ,<br><code>std::unreachable</code> , <code>std::byteswap</code>                                                    | Deducing <code>this</code> (great for<br>CRTP replacement, check<br>MSVC version)         | <code>std::shared_ptr</code> in hot<br>paths                     |

**Pinned tool versions** live in the repository root in `TOOLCHAIN.md`: Visual Studio version, Premake version, clang-format version. A formatter upgrade is its own commit.

# 03

## Repository and project layout

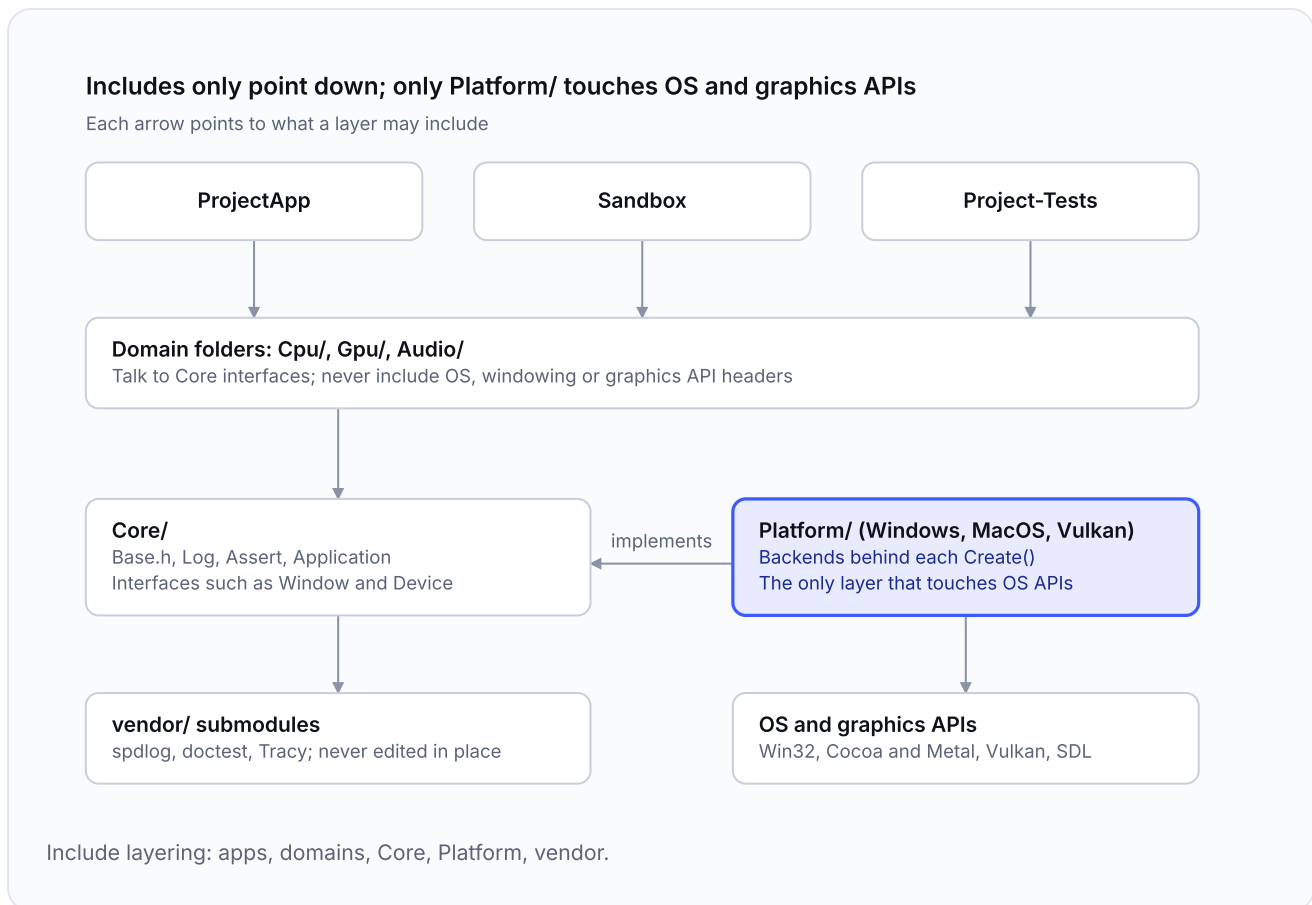
Every repository has the same shape: one engine or core library, one or more thin applications on top, vendored dependencies pulled in as submodules, and Premake scripts at the root. Tests live in the same folder as the code they test, named `\.test.cpp`, never in a parallel tree.

```
Project/
├─ premake5.lua           # workspace: configurations, platforms, startproject
├─ Dependencies.lua       # IncludeDir / Library tables for every vendor lib
├─ TOOLCHAIN.md
├─ .clang-format
├─ .clang-tidy
├─ scripts/              # Setup.py, GenerateProjects.bat/.sh, CI helpers
├─ docs/                 # architecture notes, ADRs
├─ Project/              # core library (static lib)
│   └─ premake5.lua
│   └─ src/
│       ├── prjpch.h / prjpch.cpp
│       ├── Project.h      # public umbrella header for apps
│       └─ Project/
│           ├── Core/      # Base.h, Log, Assert, Application, Timestep
│           ├── Cpu/       # domain folders, one per subsystem
│           ├── Gpu/
│           │   ├── Shader.h
│           │   ├── Shader.cpp
│           │   └─ Shader.test.cpp
│           └─ Platform/   # the only place OS and graphics APIs appear
│               ├── Windows/
│               ├── MacOS/
│               └─ Vulkan/
└─ vendor/               # git submodules, never edited in place
├─ ProjectApp/           # the shipping executable
└─ Sandbox/              # scratch app for experiments, never shipped
```

### Rules

- **MUST** keep one class per header/source pair, file named after the class: `ShaderCache.h`, `ShaderCache.cpp`.
- **MUST** put OS, windowing and graphics API includes (`<Windows.h>`, `<vulkan/vulkan.h>`, Metal, SDL) only under `Platform/`. Everything else talks to an interface.

- **MUST** include project headers by their path from `src/`: `#include "Project/Gpu/Shader.h"`, never `"../Gpu/Shader.h"`.
- **SHOULD** split a domain folder once it passes about 15 files, by sub-feature rather than by kind ( `Gpu/Pipeline/`, not `Gpu/Structs/` ).
- **SHOULD** keep `Sandbox` compiling. It is where new subsystems get exercised before an app depends on them.
- **AVOID** `Utils/`, `Helpers/` or `Common/` folders. A helper belongs to the domain that needs it; move it to `Core/` only when three domains need it.
- **MUST NOT** modify files under `vendor/`. Patch through a fork, a wrapper, or a Premake define, and note it in `Dependencies.lua`.



If a domain file needs something only an OS API can do, add a method to a Core interface and implement it in Platform/.

# 04

## Project short code and the base header

Every project picks a 2 or 3 letter uppercase short code on day one, and every macro, define and precompiled header carries it. The code keeps macros from colliding across projects and vendored libraries, and makes ownership obvious in a stack trace or a grep.

| PROJECT NAME                       | CODE | EXAMPLE MACRO       | PCH      |
|------------------------------------|------|---------------------|----------|
| Project (this guide's placeholder) | PRJ  | PRJ_CORE_ASSERT     | prjpch.h |
| Render Engine                      | RE   | RE_PLATFORM_WINDOWS | repch.h  |
| Packet Tracer                      | PT   | PT_LOG_INFO         | ptpch.h  |

### Rules

- MUST** prefix every macro with the code: `PRJ_ASSERT`, `PRJ_PLATFORM_MACOS`, `PRJ_BIT`. Unprefixed macros are allowed only as `#undef`-ed locals inside a single `.cpp`.
- MUST** name the precompiled header after the code: `prjpch.h` / `prjpch.cpp`.
- MUST** use the full project name, not the code, as the root namespace: `namespace Project`. Codes are for the preprocessor only.
- MUST** keep `Core/Base.h` as the single header that defines platform detection, build configuration, compiler wrappers, and the `Scope` / `Ref` aliases. Every other header can rely on it through the pch.
- SHOULD** prefer `constexpr` variables, `inline` functions and templates over macros. A macro earns its place only when it needs `__FILE__` / `__LINE__`, stringizing, or must vanish in a configuration.

### Base.h template

```
#pragma once

#include <memory>

// ---- Platform -----
#if defined(_WIN64)
# define PRJ_PLATFORM_WINDOWS
#elif defined(__APPLE__) && defined(__aarch64__)
# define PRJ_PLATFORM_MACOS
#elif defined(__linux__)
```

```

# define PRJ_PLATFORM_LINUX
#else
# error "Unsupported platform"
#endif

// ---- Configuration (premake sets PRJ_DEBUG / PRJ_RELEASE / PRJ_DIST) -----
#if defined(PRJ_DEBUG) || defined(PRJ_RELEASE)
# define PRJ_ENABLE_ASSERTS 1
#else
# define PRJ_ENABLE_ASSERTS 0
#endif

// ---- Compiler wrappers -----
#if defined(_MSC_VER)
# define PRJ_DEBUGBREAK() __debugbreak()
# define PRJ_FORCEINLINE __forceinline
# define PRJ_NOINLINE __declspec(noinline)
#else
# define PRJ_DEBUGBREAK() __builtin_trap()
# define PRJ_FORCEINLINE inline __attribute__((always_inline))
# define PRJ_NOINLINE __attribute__((noinline))
#endif

#define PRJ_EXPAND(x) x
#define PRJ_STRINGIFY(x) #x
#define PRJ_BIT(x) (1u << (x))

namespace Project {

    template <typename T>
    using Scope = std::unique_ptr<T>;

    template <typename T, typename... Args>
    [[nodiscard]] constexpr Scope<T> CreateScope(Args&&... args) {
        return std::make_unique<T>(std::forward<Args>(args)...);
    }

    template <typename T>
    using Ref = std::shared_ptr<T>;

    template <typename T, typename... Args>
    [[nodiscard]] constexpr Ref<T> CreateRef(Args&&... args) {
        return std::make_shared<T>(std::forward<Args>(args)...);
    }

} // namespace Project

#include "Project/Core/Assert.h"
#include "Project/Core/Log.h"

```

# 05

## Naming

---

Names follow one scheme: PascalCase for anything you can call or instantiate, camelCase for locals and parameters, and a one-letter prefix that says where a variable lives. A reader should know a name's kind and scope without hovering it.

| KIND                          | STYLE                                                                       | EXAMPLE                                                            |
|-------------------------------|-----------------------------------------------------------------------------|--------------------------------------------------------------------|
| Namespace                     | PascalCase, project name at the root                                        | <code>Project::Gpu</code>                                          |
| Class, struct, union          | PascalCase noun                                                             | <code>CommandBuffer</code> , <code>FrameStats</code>               |
| Interface (abstract base)     | PascalCase, no <code>I</code> prefix                                        | <code>Renderer</code> , implemented by <code>VulkanRenderer</code> |
| Enum type                     | PascalCase, always <code>enum class</code>                                  | <code>enum class ShaderStage : uint8_t</code>                      |
| Enumerator                    | PascalCase                                                                  | <code>ShaderStage::Fragment</code>                                 |
| Function and method           | PascalCase verb phrase                                                      | <code>LoadModule()</code> , <code>GetWidth()</code>                |
| Boolean query                 | <code>Is</code> / <code>Has</code> / <code>Can</code> / <code>Should</code> | <code>IsRunning()</code> , <code>HasDepth()</code>                 |
| Local variable, parameter     | camelCase                                                                   | <code>frameIndex</code> , <code>outBuffer</code>                   |
| Non-static data member        | <code>m_</code> + PascalCase                                                | <code>m_Width</code> , <code>m_Device</code>                       |
| Static data member            | <code>s_</code> + PascalCase                                                | <code>s_Instance</code>                                            |
| File-scope or global variable | <code>g_</code> + PascalCase, and rare                                      | <code>g_Allocator</code>                                           |
| Compile-time constant         | PascalCase, <code>constexpr</code>                                          | <code>constexpr uint32_t MaxFramesInFlight = 3;</code>             |
| Template parameter            | <code>T</code> or <code>T</code> + PascalCase                               | <code>T</code> , <code>TAllocator</code> , <code>TCallback</code>  |

| KIND                    | STYLE                            | EXAMPLE                                                               |
|-------------------------|----------------------------------|-----------------------------------------------------------------------|
| Concept                 | PascalCase adjective             | <code>Hashable</code> , <code>TriviallyRelocatable</code>             |
| Macro                   | <code>CODE_</code> + UPPER_SNAKE | <code>PRJ_CORE_ASSERT</code>                                          |
| Source files            | PascalCase, matches the class    | <code>CommandBuffer.h</code> ,<br><code>CommandBuffer.test.cpp</code> |
| Plain-data struct field | PascalCase, no <code>m_</code>   | <code>struct Extent { uint32_t Width,<br/>Height; };</code>           |

## Rules

- **MUST** keep `m_` and `s_` off public fields of plain-data structs. The prefix means "guarded by this class's invariants"; a POD has none.
- **SHOULD** name output parameters `out` + Name ( `outPixels` ) on the rare occasions they are used (see section 10).
- **SHOULD** pick getter names without `Get` when the value is cheap and obvious only if the whole class does the same. Mixed styles inside one class are not allowed.
- **AVOID** encoding types in names ( `pBuffer` , `strName` , `uiCount` ). The `m_` / `s_` / `g_` scope prefix is the only prefix.
- **AVOID** names that differ only in case or a trailing digit ( `Buffer` vs `buffer` , `pass2` ).
- **MUST** match vendor naming at the boundary only. A wrapper around `VkImage` is `Image` with a method `GetHandle()` , not `vk_image_wrapper` .

## Choosing names

A name should make its purpose clear to a new reader, even one from outside the project. Do not save horizontal space at the cost of that clarity.

- **SHOULD** make a name as descriptive as its scope is wide. A free function in a public header carries its domain in the name; a counter in a five-line loop can be `n` .
- **SHOULD NOT** repeat what the surrounding context already says. A local does not restate its function's name, and a `Swapchain` member is `m_Images` , not `m_SwapchainImages` .
- **AVOID** abbreviations a reader outside the project would not know, especially acronyms. An abbreviation is probably fine if Wikipedia lists it ( `fqdn` , `rgba` , `dma` ) or it is universal ( `i` for a loop index, `T` for a template parameter).
- **MUST NOT** abbreviate by deleting letters inside a word: `customerId` , not `cstmrId` ; `buffer` , not `bfr` .
- **MUST** capitalize an abbreviation as one word: `LoadDdsFile()` , `HttpClient` , `GpuError` , not `LoadDDSFile()` . Vendor names keep their own spelling ( `VkSwapchainKHR` ).

- **SHOULD** treat a handful of widely used names as project vocabulary, documented once in **Core/**: **Scope**, **Ref**, **Handle**, plus the short forms **dt**, **ctx**, **cmd**, **desc**, **info** and **fn**. Adding a new vocabulary name deserves a second look.

```
int CountFailedUploads(std::span<const Upload> uploads) {
    int n = 0; // clear: tiny scope, obvious meaning
    for (const auto& upload : uploads) {
        if (upload.Failed) {
            ++n;
        }
    }
    return n;
}
```

```
int CountFailedUploads(std::span<const Upload> uploads) {
    int totalNumberOfFailedUploads = 0; // Bad: too long for its scope
    for (size_t uploadIdx = 0; uploadIdx < uploads.size(); ++uploadIdx) {
        // Bad: use i for a plain index loop
    }
    int cstmrId = 0; // Bad: letters deleted from a word
    return totalNumberOfFailedUploads;
}
```

# 06

## Formatting

---

clang-format decides formatting, and nobody argues with it in review. The checked-in `.clang-format` (appendix A) encodes the choices below; run it on save and in CI.

| TOPIC                        | RULE                                                                                                                                                                                                                      |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Indentation                  | 2 spaces, never tabs. Namespace contents are indented one level.                                                                                                                                                          |
| Line length                  | 80 columns maximum (exceptions below).                                                                                                                                                                                    |
| Braces                       | Opening brace on the same line as the class, function or statement: <code>class A {</code> , <code>void Run() {</code> , <code>if (x) {</code> . Closing brace on its own line, with <code>} else {</code> kept together. |
| Access specifiers            | Indented 1 space inside the class; members indented 2.                                                                                                                                                                    |
| Pointers and references      | Attached to the type: <code>int* ptr</code> , <code>const Image&amp; image</code> .                                                                                                                                       |
| <code>const</code> placement | West const: <code>const Foo&amp;</code> , not <code>Foo const&amp;</code> .                                                                                                                                               |
| Blank lines                  | One between functions, never two. One between logical groups inside a long function.                                                                                                                                      |
| Trailing comments            | Two spaces before <code>// : }</code> <code>// namespace Project</code> .                                                                                                                                                 |
| Includes                     | Sorted and grouped by clang-format (see section 7).                                                                                                                                                                       |
| Trailing return types        | Only where needed (lambdas returning references, decltype-dependent returns).                                                                                                                                             |

- **MUST** close every namespace with a `// namespace X` comment.
- **MUST** use `// clang-format off` only around tables of data (lookup arrays, opcode maps), and switch it back on immediately after.
- **AVOID** column-aligning assignments or comments by hand; one rename breaks the alignment and the diff.

## Line length

Every line is at most 80 characters. A line may run longer only when it is:

- a comment line that cannot be split without hurting readability or copy-paste, such as one holding a long URL or an example command;
- a string literal that cannot sensibly be wrapped, such as a URL, an embedded language, or multi-line text whose newlines matter. Outside test code, put such literals at namespace scope near the top of the file, and turn clang-format off around them if it fights them;
- an `#include` line;
- a using-declaration.

## Class format

Sections go in `public`, `protected`, `private` order, each keyword indented 1 space.

```
namespace Project::Gpu {  
  
    class Swapchain final : public Presentable {  
    public:  
        explicit Swapchain(const SwapchainDesc& desc);  
        ~Swapchain() override;  
  
        [[nodiscard]] uint32_t GetImageCount() const { return m_ImageCount; }  
        bool Present(uint32_t imageIndex, std::span<const Semaphore> waitOn);  
  
    private:  
        void Recreate();  
  
        VkSwapchainKHR m_Handle = VK_NULL_HANDLE;  
        uint32_t m_ImageCount = 0;  
    };  
  
} // namespace Project::Gpu
```

- **MUST** put the base class on the same line as the class name, within the 80-column limit.
- **MUST** indent `public:`, `protected:` and `private:` 1 space, and omit any section that would be empty.
- **SHOULD** put a blank line before each access keyword except the first. Small classes may skip it.
- **MUST NOT** leave a blank line after an access keyword.

## Constructor initializer lists

An initializer list is either all on one line or wrapped before the colon and indented 4 spaces.

```
// Everything fits on one line.
Texture::Texture(Device& device) : m_Device(device) {}

// Signature and list do not fit together: wrap before the colon, indent 4.
Texture::Texture(Device& device, const TextureDesc& desc)
    : m_Device(device), m_Extent{desc.Width, desc.Height} {
    Upload(desc.Pixels);
}

// The list itself spans lines: one member per line, aligned.
Swapchain::Swapchain(const SwapchainDesc& desc)
    : m_Device(desc.Device),
      m_Surface(desc.Surface),
      m_ImageCount(desc.ImageCount) {
    Recreate();
}
```

The closing brace may sit on the same line as the opening one when the body is empty and it fits: `m_Device(device) {}`.

## Looping and branching statements

The keyword, the parenthesized condition and the opening brace sit on one line, separated by single spaces.

- MUST** put one space after `if`, `for`, `while` and `switch`, and one before `{`.
- MUST NOT** put spaces just inside the parentheses. Inside a `for` header, put one space after each semicolon.
- MUST** brace every controlled statement, with a line break after `{` and before `}`. An empty body may be `{}` on one line.
- MUST** keep `} else {` and `} else if (...) {` on the closing-brace line.
- SHOULD** declare variables needed only by the statement in its init clause: `if (auto it = m_Cache.find(key); it != m_Cache.end()) {`.
- MUST** write an empty loop body as `{}` or `continue;`, never a lone `;`.
- MAY** brace individual `case` blocks; when braced, the brace follows the label and the body is indented one more level.

```
if (frame.IsReady()) {
    Submit(frame);
} else if (auto* fence = m_Fences.Find(frame.Index); fence != nullptr) {
    fence->Wait();
} else {
    Skip(frame);
}
```

```

for (size_t i = 0; i < count; ++i) {
    Upload(i);
}

switch (stage) {
    case ShaderStage::Vertex: {
        CompileVertex(source);
        break;
    }
    case ShaderStage::Fragment:
        CompileFragment(source);
        break;
}

while (TryPop(item)) {} // empty body: braces, never a lone semicolon

```

```

if(ready){ Submit(frame); } // Bad: missing spaces
if ( ready ) { // Bad: spaces inside the parentheses
if (ready) Submit(frame); // Bad: every controlled statement is braced
while (TryPop(item)); // Bad: reads like the tail of a do-while

```

The Google guide tolerates unbraced one-line bodies but discourages them and lets projects require braces; these projects do.

## Lambda formatting

Format a lambda's parameters and body like any other function, and its capture list like any other comma-separated list.

- **MUST NOT** put a space between `&` and the name in a by-reference capture: `[&total]`.
- **SHOULD** write short lambdas inline as arguments.

```

auto addTo = [&total](int value) { total += value; };

std::erase_if(m_Pending, [&retired](const Job& job) {
    return retired.contains(job.Id);
});

```

## Pointer and reference expressions

- **MUST NOT** put spaces around `.` or `->`, or after a unary `*` or `&`: `x = *ptr;`, `ptr = &x;`, `v = obj->Field;`.
- **MUST** attach `*` and `&` to the type in declarations, with one space before the name: `char* name`, `const Image& image`, `int* GetPointer();`. No space before a closing `>`: `std::vector<char*>`.

- **MAY** declare several variables in one statement only when none of them has `*` or `&`: `int x, y;` is fine, `int x, *y;` is not.

## Return values

- **MUST NOT** wrap a `return` expression in parentheses unless you would also use them in `x = expr;`. Write `return result;`, never `return (result);` or `return(result);`.
- **MAY** parenthesize a long expression that is broken across lines, where the parentheses help the reader see where it ends.

## Preprocessor directives

The `#` of a directive always starts at column 0, even inside indented code. Show nesting with spaces after the `#`.

```
void Device::Submit() {
    Flush();
    #if PRJ_ENABLE_GPU_VALIDATION
    #   if defined(PRJ_PLATFORM_WINDOWS)
        EnableDebugLayer();
    #   endif
    #endif
    QueueSubmit();
}
```

## Floating-point literals

- **MUST** write a radix point with digits on both sides, even with an exponent: `1.0f`, `-0.5`, `1248.0e6`. Never `1.f`, `-.5` or `1248e6`.
- **MAY** initialize a floating-point variable from an integer literal the type can represent exactly: `float scale = 1;`

## Non-ASCII characters

- **SHOULD** keep non-ASCII characters rare. User-facing text does not belong in source; parsers of foreign data formats and test strings are the usual exceptions.
- **MUST** encode source files as UTF-8 (MSVC builds with `/utf-8`).
- **SHOULD** use escapes for invisible or confusable characters: `"`, not a literal byte-order mark.
- **AVOID** the `u8` prefix; its meaning changed in C++20 (`char8_t`).
- **MUST NOT** use `char16_t` or `char32_t`. Use `wchar_t` only at Windows API calls inside `Platform/Windows/`, converting to UTF-8 `std::string` right there.

# 07

## Headers and includes

---

Headers declare, sources define, and every header compiles on its own. Build times are a feature: a header that drags in `<Windows.h>` or a vendor SDK costs every file that includes it.

**Include order** in a `.cpp` (clang-format enforces the groups, separated by one blank line):

1. The precompiled header: `#include "prjpch.h"` (first line of every `.cpp`, MSVC requires it).
2. The file's own header: `#include "Project/Gpu/Shader.h"`.
3. Other project headers.
4. Vendor headers in angle brackets: `<spdlog/spdlog.h>`, `<vulkan/vulkan.h>`.
5. Standard library headers: `<vector>`, `<span>`.

Headers follow the same order minus the pch.

### Rules

- **MUST** start every header with `#pragma once`. No include guards.
- **MUST** make every header self-sufficient: a `.cpp` containing only that include must compile. The `.test.cpp` next to it proves this for free.
- **MUST** keep `using namespace` out of headers entirely. In a `.cpp`, only inside a function or for `std::literals`.
- **SHOULD** forward-declare instead of including when a header only uses a pointer, reference or `Scope<T>` member of the type (the destructor then goes in the `.cpp`).
- **SHOULD** keep platform and vendor headers out of public headers. Hide them behind PIMPL or an opaque handle ( `using NativeHandle = void*;` ) when an interface must expose them.
- **SHOULD** put into the pch only what is stable and widely used: the standard library, `Base.h`, `Log.h`, and at most one heavy platform header per platform. Never put a project header that changes weekly in it.
- **AVOID** umbrella headers inside the library. `Project.h` exists for apps; library code includes exactly what it uses.
- **AVOID** C++20 modules for now. Revisit once Premake, MSVC and Clang agree on BMI handling; record the decision in an ADR when it changes.

```

// Gpu/Texture.h
#pragma once

#include "Project/Core/Base.h"

#include <span>

namespace Project::Gpu {

    class Device;          // forward declaration: only used by reference
    struct TextureImpl;    // PIMPL hides the Vulkan types

    class Texture {
    public:
        Texture(Device& device, std::span<const std::byte> pixels);
        ~Texture();        // defined in Texture.cpp, where TextureImpl is complete

    private:
        Scope<TextureImpl> m_Impl;
    };

} // namespace Project::Gpu

```

# 08

## Types, ownership and lifetimes

Ownership is visible in the type: `Scope<T>` owns alone, `Ref<T>` shares, a raw pointer or reference never owns. If a reader has to read a function body to learn who frees something, the signature is wrong.

| YOU HAVE                                                              | USE                                                                                          | NEVER                                                                                               |
|-----------------------------------------------------------------------|----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| Sole owner of a heap object                                           | <code>Scope&lt;T&gt;</code> via <code>CreateScope&lt;T&gt;()</code>                          | <code>new</code> / <code>delete</code> in application code                                          |
| Genuinely shared lifetime (assets, resources held by several systems) | <code>Ref&lt;T&gt;</code> via <code>CreateRef&lt;T&gt;()</code>                              | <code>Ref&lt;T&gt;</code> "just in case"                                                            |
| Non-owning, never null                                                | <code>T&amp;</code> / <code>const T&amp;</code>                                              | Pointer that is "never null" by convention                                                          |
| Non-owning, may be null                                               | <code>T*</code>                                                                              | <code>Ref&lt;T&gt;</code> passed by value to avoid thinking                                         |
| Optional value                                                        | <code>std::optional&lt;T&gt;</code>                                                          | Sentinel values like <code>-1</code> or <code>nullptr</code> for "none"                             |
| Contiguous view                                                       | <code>std::span&lt;T&gt;</code> / <code>std::span&lt;const T&gt;</code>                      | Pointer + size pairs                                                                                |
| String view                                                           | <code>std::string_view</code> (not stored, not passed to C APIs needing <code>\0</code> )    | <code>const char*</code> outside C ABI boundaries                                                   |
| Handle to a pooled object                                             | Strong typed index: <code>struct MeshHandle { uint32_t Index; uint32_t Generation; };</code> | Raw pointers into a vector that may reallocate                                                      |
| Fixed-size integer                                                    | <code>uint8_t</code> ... <code>int64_t</code> , <code>size_t</code> for sizes and indices    | <code>int</code> / <code>long</code> for anything with a defined width; <code>unsigned</code> alone |
| Bytes                                                                 | <code>std::byte</code> and <code>std::span&lt;std::byte&gt;</code>                           | <code>char*</code> / <code>unsigned char*</code> for raw memory                                     |

## Rules

- **MUST NOT** write `new` or `delete` outside allocator and container implementations.
- **MUST** use `enum class` with an explicit underlying type for anything stored or serialized.
- **SHOULD** wrap meaningful numbers in small strong types once they cross a module boundary: `struct Address { uint32_t Value; };` stops a size being passed as an address. Emulators and RE tools lean on this heavily.
- **SHOULD** use `auto` when the type is on the same line ( `auto device = CreateScope<Device>()` ), for iterators, and for lambdas. Spell the type when it carries meaning a reader needs ( `uint32_t` , `float` ).
- **SHOULD** use `std::string` for owned strings and `std::string_view` for read-only parameters. Convert to `std::string` before handing to any API that needs null termination.
- **AVOID** `std::shared_ptr` cycles; break them with `std::weak_ptr` or an owning parent. If two `Ref` s point at each other, redesign.
- **AVOID** `reinterpret_cast` and C-style casts. For byte reinterpretation use `std::bit_cast` ; for game memory structures, use explicit offsets and `std::memcpy` .

**Lifetime traps to catch in review:** a `string_view` or `span` outliving its source, a reference captured by a lambda queued for later, an iterator held across `push_back` , and `this` captured into a callback that outlives the object.

# 09

## Classes and interfaces

---

A `struct` is data, a `class` is behavior that protects an invariant. Classes are small, final by default, and own their resources through members that clean up after themselves, so most of them need no special member functions at all.

### struct or class

- **MUST** use `struct` only for passive objects that carry data. Every field is public, and no rule ties one field to another, because callers can change any field directly.
- **MAY** give a struct constructors, a destructor or helper methods, as long as none of them requires or enforces an invariant. Build structs with designated initializers: `TextureDesc{.Width = 256, .Format = Format::Rgba8}`.
- **MUST** use `class` once a type needs an invariant or more behavior, or once a widely used struct is expected to grow. When in doubt, make it a `class`.
- **MAY** use `struct` for stateless types such as traits, template metafunctions and small functors, to match the standard library.

**Member order** applies within each of the public, protected and private sections, in this sequence:

1. Types and type aliases ( `using` , nested enums, structs and classes, `friend` types)
2. Non-static data members (structs only)
3. Static constants
4. Factory functions ( `Create` )
5. Constructors and assignment operators
6. Destructor
7. All other functions, static and non-static
8. All other data members, last in the `private` section

### Special members

- **SHOULD** follow the rule of zero: let `Scope` , `std::vector` and RAII wrappers handle cleanup so the compiler-generated members are right.
- **MUST** follow the rule of five when a class owns a raw handle ( `VkDevice` , `HANDLE` , a file descriptor): declare destructor, move constructor and move assignment; delete copy unless copying is meaningful.

- **MUST** mark single-argument constructors and conversion operators `explicit`.
- **SHOULD** mark classes `final` unless designed for inheritance. Base classes intended for inheritance get a `virtual` destructor or a `protected` non-virtual one.

## Interfaces

Platform-dependent systems expose an abstract interface with a static `Create` that picks the backend. The backend lives under `Platform/` and is never named outside it.

```
// Project/Core/Window.h
class Window {
public:
    static Scope<Window> Create(const WindowDesc& desc);

    virtual ~Window() = default;

    [[nodiscard]] virtual uint32_t GetWidth() const = 0;
    virtual void PollEvents() = 0;
};

// Project/Platform/Windows/WindowsWindow.cpp
Scope<Window> Window::Create(const WindowDesc& desc) {
    return CreateScope<WindowsWindow>(desc);
}
```

- **MUST** mark every override `override`, never repeat `virtual` on it.
- **SHOULD** choose a compile-time backend (one `Create` definition per platform, selected by Premake `filter`) over a runtime switch when only one backend can exist per build.
- **AVOID** deep inheritance. Two levels (interface, implementation) is the norm; a third needs a reason.
- **AVOID** singletons. When one is unavoidable ( `Application` ), expose it through a static `Get()` that asserts it exists, and never construct it lazily from an arbitrary thread.
- **AVOID** `friend` except for tests and tightly paired types in the same file.

# 10

## Functions

---

Functions are short, do one thing, and return their results instead of writing through parameters. A function that needs a comment to explain its sections should be several functions.

### Passing parameters

| PARAMETER IS                                                                              | PASS AS                                                                                     |
|-------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| Cheap to copy ( $\leq 16$ bytes, trivially copyable: ints, handles, <code>Extent</code> ) | By value                                                                                    |
| Read-only and larger                                                                      | <code>const T&amp;</code>                                                                   |
| A string you only read                                                                    | <code>std::string_view</code>                                                               |
| A range you only read                                                                     | <code>std::span&lt;const T&gt;</code>                                                       |
| Consumed or stored by the callee                                                          | By value, then <code>std::move</code> into place                                            |
| Modified in place (rare)                                                                  | <code>T&amp;</code> , named so the call site reads as mutation                              |
| A callback invoked during the call only                                                   | Template parameter constrained by <code>std::invocable</code> , or <code>FunctionRef</code> |
| A callback stored for later                                                               | <code>std::function</code> or <code>std::move_only_function</code>                          |

### Rules

- **SHOULD** return values, structs or `std::expected` rather than out-parameters. Return value optimization makes this free.
- **MUST** mark functions `[[nodiscard]]` when ignoring the result is a bug: anything returning `std::expected`, a handle, an owning pointer, or a computed value with no side effects.
- **MUST** mark functions `noexcept` when they cannot throw and sit on a hot path or in a move operation.
- **SHOULD** keep to about 4 parameters. Past 4, take a `*Desc` struct with designated initializers.

- **AVOID** `bool` parameters at call sites. `Upload(data, true)` says nothing; use an `enum class` or two named functions.
- **AVOID** default arguments on virtual functions; the default comes from the static type.
- **SHOULD** prefer early returns over nested `if` blocks. Guard clauses first, happy path last, unindented.

```
// Good: guard clauses, a Desc struct, an expected result.
[[nodiscard]] std::expected<Scope<Texture>, GpuError> LoadTexture(
    Device& device, const TextureLoadDesc& desc) {
    if (desc.Path.empty()) {
        return std::unexpected(GpuError::InvalidArgument);
    }

    auto pixels = ReadImageFile(desc.Path);
    if (!pixels) {
        return std::unexpected(pixels.error());
    }

    return CreateScope<Texture>(device, *pixels);
}
```

## Write short functions

Prefer small, focused functions. There is no hard limit, but once a function passes about 40 lines, ask whether it can be split without hurting the program's structure.

A long function that works today is where tomorrow's change hides a bug. Short functions are easier to read, change and test. When an existing long function is hard to modify or debug, or you want to reuse part of it, split it rather than working around it.

## Local variables

Declare each variable in the narrowest scope possible, as close to its first use as possible, and initialize it where it is declared.

- **MUST** initialize in the declaration rather than declaring and then assigning: `int jobs = CountJobs();`, not `int jobs; jobs = CountJobs();`.
- **SHOULD** start containers initialized: `std::vector<int> ids = {1, 2};` rather than a series of `push_back` calls.
- **SHOULD** declare variables used only by an `if`, `while` or `for` inside that statement, so they cannot leak past it.
- **SHOULD** hoist an object out of a loop when constructing and destroying it every iteration is costly, and reuse it instead.

```

int jobs = CountJobs();
ScheduleJobs(jobs); // declaration right before use

while (const char* slash = std::strchr(path, '/')) {
    path = slash + 1;
}

ShaderCompiler compiler; // built once, not once per source
for (const auto& source : sources) {
    compiler.Compile(source);
}

```

## Lambda expressions

- **SHOULD** use lambdas where they make code clearer, formatted as in section 6.
- **MUST** capture explicitly when the lambda may escape the current scope (stored, queued, or run on another thread). A default capture hides that a reference, or `this`, went along.
- **MAY** use `[&]` only when the lambda's lifetime is obviously shorter than anything it could capture, such as an algorithm predicate or an immediate visitor.
- **MAY** use `[=]` only for a short lambda that binds a few obvious variables, and never where it would capture `this` implicitly. A lambda in a member function that uses members captures `this` explicitly.
- **MUST** use init captures only to capture, such as moving an owner in: `[texture = std::move(texture)]`. Do not use them to invent new names or change what an existing name means; declare a variable first, or write a named function object.
- **MUST NOT** capture references into a lambda that is queued or run on another thread. Capture by value, or capture a `std::weak_ptr`.
- **SHOULD** name a lambda (`const auto`) once it passes about 5 lines or is used twice.

```

// Bad: [&] hides that `scene` and `this` are captured; if the job runs
// after this function returns, both may be gone.
m_Jobs.Submit([&] { Rebuild(scene); });

// Better: the captures are visible, and the job owns its scene copy.
m_Jobs.Submit([this, snapshot = scene.Snapshot()] { Rebuild(snapshot); });

```

# 11

## Error handling

Errors split into two kinds, and each kind has one tool. A programmer mistake (a broken invariant) is an assert and stops the program in development. An expected runtime failure (a missing file, a lost device, a bad packet) is a `std::expected` the caller must handle.

| SITUATION                                                                                 | TOOL                                                                            | SURVIVES INTO DIST?                     |
|-------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|-----------------------------------------|
| Invariant that can only fail through a bug                                                | <code>PRJ_ASSERT(cond, msg)</code> / <code>PRJ_CORE_ASSERT</code>               | No, compiled out                        |
| Invariant whose expression has a needed side effect, or that guards memory safety         | <code>PRJ_VERIFY(cond, msg)</code>                                              | Yes, logs and breaks or aborts          |
| Code path that must be unreachable                                                        | <code>PRJ_UNREACHABLE()</code> (asserts, then <code>std::unreachable()</code> ) | Yes, as <code>std::unreachable()</code> |
| Recoverable failure the caller should decide about                                        | <code>std::expected&lt;T, Error&gt;</code>                                      | Yes                                     |
| Value that may legitimately be absent                                                     | <code>std::optional&lt;T&gt;</code>                                             | Yes                                     |
| Failure crossing a C ABI or DLL boundary                                                  | Enum result code + out-pointer                                                  | Yes                                     |
| Truly unrecoverable environment failure (out of memory, GPU device lost with no recovery) | Log, then <code>PRJ_FATAL</code> which flushes logs and terminates              | Yes                                     |

### Exceptions

- **MUST NOT** throw exceptions in project code. Projects build with exceptions enabled only so vendor libraries that need them work.
- **MUST** catch at the boundary any exception a vendor library can throw, and convert it to an `Error` value right there.
- **MUST NOT** let any exception cross a DLL, C ABI, script VM or thread boundary.

### Error types

- **SHOULD** give each subsystem one small `enum class` error ( `GpuError` , `IoError` ) plus, when context matters, a struct carrying the enum and a message.
- **MUST** propagate errors with early returns; an `expected` is never silently dropped (the `[[nodiscard]]` rule enforces this).
- **SHOULD** log an error once, at the place that decides what to do about it, not at every level it passes through.

```
enum class IoError : uint8_t { NotFound, AccessDenied, Corrupt };

template <typename T>
using IoResult = std::expected<T, IoError>;

[[nodiscard]] IoResult<std::vector<std::byte>> ReadFile(
    const std::filesystem::path& path);

// At the decision point:
auto bytes = ReadFile(romPath);
if (!bytes) {
    PRJ_CORE_ERROR("Could not load {}: {}", romPath.string(),
        ToString(bytes.error()));
    return false;
}
```

# 12

## const and compile-time code

---

Everything is `const` until it needs to change, and anything that can be computed at compile time is. Mutability and runtime work should be the visible exceptions.

- **MUST** mark member functions `const` when they do not change observable state. Use `mutable` only for caches and mutexes, with a comment.
- **SHOULD** declare locals `const` when they are not reassigned. Parameters passed by value are not marked `const` in declarations (it is noise in the header) but may be in definitions.
- **MUST** use `constexpr` for constants, never `#define` or `static const` in headers.
- **SHOULD** make lookup tables, hash seeds, bit masks and format descriptors `constexpr` (or `constexpr`-built) so they live in read-only data with no static initialization order risk.
- **SHOULD** use `constexpr` for functions that must never run at runtime: compile-time string hashing, opcode table generation, validated format strings.
- **SHOULD** use `if constexpr` instead of tag dispatch or SFINAE when a template branches on a type property.
- **SHOULD** `static_assert` every layout assumption: struct sizes and offsets that match hardware, file formats or another process's memory.
- **AVOID** non-`constexpr` globals with dynamic initialization. When a global is unavoidable, use a function-local static or construct it explicitly at startup.

```
// Opcode decode table built entirely at compile time.
constexpr std::array<OpHandler, 64> BuildPrimaryTable() {
    std::array<OpHandler, 64> table{};
    table.fill(&Interpreter::Unimplemented);
    table[14] = &Interpreter::Addi;
    table[15] = &Interpreter::Addis;
    return table;
}

inline constexpr auto PrimaryTable = BuildPrimaryTable();

// Layout that must match guest memory exactly.
struct GuestThreadContext {
    uint64_t Gpr[32];
    uint64_t Lr;
    uint64_t Ctr;
};
```

```
static_assert(sizeof(GuestThreadContext) == 0x110);  
static_assert(offsetof(GuestThreadContext, Lr) == 0x100);
```

# 13

## Templates and concepts

---

Templates are for code that is genuinely generic, and every template parameter is constrained by a concept. An unconstrained template turns a one-line mistake into a 300-line MSVC error.

- **MUST** constrain template parameters with a concept or `requires` clause. Prefer the short form: `template <std::integral T> .`
- **SHOULD** name project concepts after what the type can do ( `Serializable` , `GpuResource` ) and keep them in the domain that uses them.
- **SHOULD** replace CRTP with deducing `this` where MSVC supports it in the pinned toolchain; keep CRTP only where it does not.
- **SHOULD** keep template definitions in the header that declares them. When a template is large and used with few types, move the body to a `.inl` or `.cpp` and add explicit instantiations.
- **AVOID** SFINAE ( `std::enable_if` ) in new code; concepts replace it.
- **AVOID** template metaprogramming that a `constexpr` function could do. Readable compile-time code beats clever type lists.
- **AVOID** templates as a way to skip writing an interface. If there will only ever be one or two types, write two functions.

```
template <typename T>
concept TriviallyRelocatable =
    std::is_trivially_copyable_v<T> && std::is_trivially_destructible_v<T>;

template <TriviallyRelocatable T>
class RingBuffer {
public:
    explicit RingBuffer(size_t capacity);

    bool Push(const T& value) noexcept;
    [[nodiscard]] std::optional<T> Pop() noexcept;

    // Deducing this: one accessor for const and non-const.
    template <typename Self>
    auto&& Front(this Self&& self) {
        return std::forward<Self>(self).m_Data[self.m_Head];
    }
}
```

```
private:  
    std::vector<T> m_Data;  
    size_t m_Head = 0;  
    size_t m_Tail = 0;  
};
```

# 14

## Memory and performance

---

Performance comes from data layout and from not allocating in the loop, and it is proven with a profiler, not argued in review. Write the clear version first, measure, then optimize the hot 5% with a comment saying what the profiler showed.

### Allocation

- **MUST NOT** allocate on the heap per frame, per tick, per packet or per emulated instruction. Hot loops use pre-sized containers, frame (linear) allocators, or pools.
- **SHOULD** `reserve()` vectors whose final size is known or bounded.
- **SHOULD** use a per-frame linear allocator for transient data (render commands, temporary strings) and reset it once per frame.
- **SHOULD** route subsystem allocations through a named allocator so the memory profiler can attribute them ( `PRJ_ALLOC_TAG("Gpu")` ).

### Containers

| NEED                        | DEFAULT                                                               | WHEN TO SWITCH                                                           |
|-----------------------------|-----------------------------------------------------------------------|--------------------------------------------------------------------------|
| Sequence                    | <code>std::vector</code>                                              | Almost never; <code>std::deque</code> only for stable references on push |
| Small fixed maximum         | <code>std::array</code> or an inline-capacity vector                  | —                                                                        |
| Key → value lookup          | Open-addressing hash map (e.g. <code>ankerl::unordered_dense</code> ) | <code>std::unordered_map</code> only when pointer stability is required  |
| Sorted keys, few inserts    | Sorted <code>std::vector</code> + <code>std::lower_bound</code>       | <code>std::map</code> only for frequent ordered inserts                  |
| Objects with stable handles | Slot map / pool with generational handles                             | —                                                                        |

### Layout and moves

- **SHOULD** store hot data as structure-of-arrays when a loop touches one or two fields of many objects (chunk voxels, particles, entity transforms).
- **SHOULD** order struct members largest to smallest to avoid padding in types stored in bulk; `static_assert` the size.
- **MUST** make move constructors and move assignment `noexcept`, or `std::vector` will copy on growth.
- **SHOULD** pass `std::move` only when the source is not used afterward, and never `return std::move(local);` (it blocks copy elision).
- **AVOID** virtual calls inside per-element loops. Batch by type, or resolve the call once outside the loop.

## Profiling

Every project wires a profiler (Tracy is the default) through `PRJ_PROFILE_FRAME()`, `PRJ_PROFILE_SCOPE(name)` and `PRJ_PROFILE_FUNCTION()`. These compile to nothing unless `PRJ_ENABLE_PROFILING` is set, so they stay in the code permanently. A performance claim in a commit message names the scenario and the before and after numbers.

# 15

## Concurrency

---

Every piece of mutable data has exactly one owning thread at any moment, and handing it to another thread is an explicit, visible step. Shared mutable state with locks is the fallback, not the design.

- **SHOULD** structure work as jobs over owned data (chunk generation, mesh building, asset decode) submitted to the project's job system, rather than ad hoc `std::thread`s.
- **MUST** name every long-lived thread ( `SetThreadName("Gpu Submit")` ) so it shows up in the debugger and profiler.
- **MUST** use `std::jthread` with a `std::stop_token` for long-lived threads; never detach.
- **MUST** guard shared data with a mutex declared next to it, and keep the lock scope as small as the critical section. Use `std::scoped_lock` for more than one mutex.
- **SHOULD** pass data between threads through queues (SPSC or MPMC ring buffers) instead of shared fields.
- **MUST** use `std::atomic` for flags and counters touched by more than one thread. Default to `memory_order_seq_cst`; relax the order only with a comment explaining why it is safe.
- **MUST** document thread affinity on any class that is not thread-safe: a one-line comment `// Main thread only.` above the class, backed by `PRJ_ASSERT_MAIN_THREAD()` in debug builds.
- **AVOID** `volatile` for synchronization. It means "memory-mapped hardware", nothing more.
- **AVOID** holding a lock while calling into code you do not own (callbacks, logging sinks, vendor APIs).

**Review checklist for threaded code:** who owns this data now, what happens when the stop token fires mid-job, what keeps captured objects alive, and does a ThreadSanitizer (Clang) run come back clean.

# 16

## Platform abstraction and API boundaries

---

Platform code is quarantined in `Platform/`, and anything that crosses a binary boundary (a DLL, a plugin, a scripting VM, an injected module) speaks plain C. Inside the project you get all of C++23; at the edge you get structs, enums and functions.

### Platform code

- **MUST** keep `#ifdef PRJ_PLATFORM_*` out of non-platform files. Select platform sources in Premake with `filter "system:windows"` instead.
- **SHOULD** give each platform folder the same file names ( `Platform/Windows/WindowsWindow.cpp` , `Platform/MacOS/MacOSWindow.mm` ) so the parallel structure is obvious.
- **SHOULD** wrap OS handles in RAII types inside `Platform/` ( `UniqueHandle` , `UniqueVkObject<T>` ) so leaks are impossible by construction.

### C ABI boundaries

- **MUST** export only `extern "C"` functions taking and returning C types: fixed-width integers, `const char*` , plain structs, opaque handles.
- **MUST** version every exported struct with a leading `uint32_t StructSize` (or a version field) so it can grow without breaking callers.
- **MUST** use a visibility macro for exports: `PRJ_API` expands to `__declspec(dllexport)` / `__declspec(dllimport)` on Windows and `__attribute__((visibility("default")))` elsewhere; build with hidden visibility by default on Clang.
- **MUST** let the module that allocates memory also free it. Export a matching `prj_free_*` for anything returned across the boundary.
- **MUST NOT** pass `std::string` , `std::vector` , smart pointers or exceptions across the boundary.
- **SHOULD** name C ABI functions in `snake_case` with the lowercase code prefix ( `prj_session_open` ), so the boundary is visible at every call site.

```
// Project/Api/ProjectApi.h: the only header external tools include.
#pragma once
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif
```

```

typedef struct PrjSession PrjSession; // opaque

typedef enum PrjResult : int32_t {
    PRJ_OK = 0,
    PRJ_ERROR_INVALID_ARGUMENT = 1,
    PRJ_ERROR_NOT_FOUND = 2,
} PrjResult;

typedef struct PrjSessionDesc {
    uint32_t StructSize;
    const char* ImagePath;
} PrjSessionDesc;

PRJ_API PrjResult prj_session_open(const PrjSessionDesc* desc,
                                   PrjSession** outSession);
PRJ_API void prj_session_close(PrjSession* session);

#ifdef __cplusplus
} // extern "C"
#endif

```

# 17

## Logging, assertions and diagnostics

---

Logging goes through two loggers: a core logger for the library and a client logger for the application, both behind prefixed macros so the backend (spdlog by default) can change in one file. What each configuration keeps is fixed:

| MACRO                                                  | DEBUG                    | RELEASE            | DIST               |
|--------------------------------------------------------|--------------------------|--------------------|--------------------|
| <code>PRJ_CORE_TRACE</code> / <code>PRJ_TRACE</code>   | Yes                      | No                 | No                 |
| <code>PRJ_CORE_INFO</code> / <code>PRJ_INFO</code>     | Yes                      | Yes                | No                 |
| <code>PRJ_CORE_WARN</code> / <code>PRJ_WARN</code>     | Yes                      | Yes                | Yes                |
| <code>PRJ_CORE_ERROR</code> / <code>PRJ_ERROR</code>   | Yes                      | Yes                | Yes                |
| <code>PRJ_CORE_FATAL</code> / <code>PRJ_FATAL</code>   | Yes, then break          | Yes, then abort    | Yes, then abort    |
| <code>PRJ_ASSERT</code> / <code>PRJ_CORE_ASSERT</code> | Yes, break into debugger | Yes, log and break | Compiled out       |
| <code>PRJ_VERIFY</code>                                | Yes, break into debugger | Yes, log and break | Yes, log and abort |
| <code>PRJ_PROFILE_*</code>                             | Opt-in                   | Opt-in             | No                 |

### Rules

- MUST** use `std::format`-style placeholders (`"{}"`), never `printf` specifiers or string concatenation.
- MUST** put `CORE` macros only in the library and plain macros only in applications. The log prefix then says which side of the line a message came from.
- MUST** give every assert a message saying what was expected, in words: `PRJ_CORE_ASSERT(index < m_Count, "Buffer index {} out of range ({})", index, m_Count)`.
- SHOULD** log at `TRACE` inside loops and only behind a category or verbosity check.
- SHOULD** log state changes and decisions ("Swapchain recreated at 2560x1440"), not function entry and exit.
- MUST NOT** log secrets, tokens, credentials or personal data, in any configuration.

- **SHOULD** install a crash handler in Release and Dist that writes a minidump (Windows) or symbolized backtrace (macOS) plus the last 200 log lines to the app's log folder.

# 18

## Testing

---

**MUST** put every test in the same folder as the code it tests, named `<name>.test.cpp` after the file under test: `Decoder.cpp` is tested by `Decoder.test.cpp` right beside it. There is no separate test tree. Premake gathers every `*.test.cpp` into one test executable per library and excludes them from the library itself.

```
Project/src/Project/Cpu/  
├─ Decoder.h  
├─ Decoder.cpp  
└─ Decoder.test.cpp
```

**Framework.** doctest is the default: header-only, fast to compile, and its `TEST_CASE` / `SUBCASE` / `CHECK` syntax reads cleanly next to source. Catch2 v3 is the alternative when a project needs its benchmarking or matchers.

### Rules

- **MUST** keep `*.test.cpp` files out of the shipping library and app targets (Premake `removefiles` on the library, `files` on the test project).
- **MUST** add a test with every bug fix that reproduces the bug first.
- **SHOULD** test through the public interface. Reaching into privates means the class wants splitting.
- **SHOULD** name test cases as behavior sentences: `TEST_CASE("Decoder rejects a branch with a misaligned target")`.
- **SHOULD** make tests deterministic: fixed seeds, no wall-clock time, no network, no GPU unless the test project is explicitly a GPU test.
- **SHOULD** use golden files for emulators, decoders and renderers (expected trace output, expected image hashes), kept in a `testdata/` folder beside the `.test.cpp` that uses them, and regenerate them only in a dedicated commit.
- **SHOULD** run tests in CI under Debug with AddressSanitizer and UndefinedBehaviorSanitizer (Clang), and under MSVC with `/fsanitize=address`.
- **AVOID** mocks for everything. Fake the platform layer at the interface (a `HeadlessWindow`, a `NullDevice`) and test real logic against it.

```

// Decoder.test.cpp
#include "prjpch.h"
#include "Project/Cpu/Decoder.h"

#include <doctest/doctest.h>

namespace Project::Cpu {

    TEST_CASE("Decoder reads addi with a negative immediate") {
        const auto instr = Decode(0x3860FFFF); // li r3, -1
        REQUIRE(instr.has_value());
        CHECK(instr->Op == Opcode::Addi);
        CHECK(instr->Rd == 3);
        CHECK(instr->Imm == -1);
    }

} // namespace Project::Cpu

```

# 19

## Build system

---

Premake generates every build: Visual Studio solutions on Windows, Xcode or gmake2 on macOS and Linux. CMake appears only when a vendored dependency forces it, and even then it is wrapped by a Premake project or prebuilt by a setup script.

**Configurations** are the same three in every project:

| CONFIGURATION | DEFINE                   | OPTIMIZATION | SYMBOLS                       | ASSERTS | USE                           |
|---------------|--------------------------|--------------|-------------------------------|---------|-------------------------------|
| Debug         | <code>PRJ_DEBUG</code>   | Off          | Full                          | On      | Daily development, sanitizers |
| Release       | <code>PRJ_RELEASE</code> | On           | Full                          | On      | Profiling, testers            |
| Dist          | <code>PRJ_DIST</code>    | On, LTO      | Stripped, PDB kept separately | Off     | What ships                    |

### Rules

- MUST** set `cppdialect "C++23"` (or `"C++latest"` on MSVC until 23 is accepted), `staticruntime "off"`, and `flags { "MultiProcessorCompile" }` in the workspace.
- MUST** treat warnings as errors ( `fatalwarnings` ) in CI on both MSVC and Clang; locally it may be off while iterating.
- MUST** keep output paths uniform: `bin/{cfg.buildcfg}-{cfg.system}-{cfg.architecture}/{prj.name}` and `bin-int/` for intermediates. Both are git-ignored.
- MUST** list every vendor library's include path and link name once, in `Dependencies.lua`, and reference them by table key.
- MUST** produce a clean MSVC build in CI on every push, even for projects developed mainly on the Mac.
- SHOULD** keep `GenerateProjects.bat` and `GenerateProjects.sh` in `scripts/` so a fresh clone is one command from a building solution.
- SHOULD** enable `/Zc:inline`, `/Zc:throwingNew` and `/utf-8` on MSVC; `-fvisibility=hidden` and `-fno-omit-frame-pointer` on Clang.

- **AVOID** per-file compiler flags. If one file needs different flags, it probably belongs in a separate vendor project.

The full workspace template is in appendix C.

# 20

## Comments and documentation

---

Comments explain why, the code explains what. A comment that restates the next line is deleted in review; a comment that records a hardware quirk, a reverse-engineered offset or a rejected alternative is gold.

### Comment style

- **MUST** write comments with `//`. Google allows `//` or `/* */` as long as a codebase is consistent; these projects pick `//` everywhere, doc comments included.
- **MUST** cite the source for magic numbers and reverse-engineered facts: `// Timeout of 250 ms, per section 4.2 of the device datasheet` or `// Offset 0x1C8 confirmed against library v1.2.3`.
- **MUST** write `TODO` and `FIXME` with an owner or issue: `// TODO(owner): handle swapchain loss during resize (#42)`. An anonymous `TODO` is noise.
- **SHOULD** use `// NOTE:` for surprising-but-correct code and `// HACK:` for knowingly wrong code with a removal plan.
- **AVOID** commented-out code. Git remembers it.
- **AVOID** banner comments and ASCII-art separators except in `Base.h`-style headers that group many macros.
- **SHOULD** record big decisions (a library choice, an allocation strategy, a threading model) as short ADRs in `docs/adr/NNNN-title.md`: context, decision, consequences.

### Struct and class comments

Every non-obvious class or struct gets a comment saying what it is for and how to use it.

- **SHOULD** give the reader enough to know when and how to use the type, plus anything needed to use it correctly.
- **MUST** state the threading assumptions. If instances can be used from several threads, spell out the rules and invariants for doing so.
- **SHOULD** include a short usage example when one makes the intent obvious.
- **SHOULD** keep comments about using the class with its declaration in the `.h`, and comments about how it works with the method definitions in the `.cpp`.

```
// Streams texture data from disk into GPU memory on the transfer queue.
// Uploads become visible after the next Device::FlushUploads().
// Main thread only: Request() and Poll() must not race.
//
// Example:
// TextureStreamer streamer(device);
// auto ticket = streamer.Request("textures/rock.dds");
// while (!streamer.Poll(ticket)) {
//     DoOtherWork();
// }
class TextureStreamer {
    ...
};
```

## Function comments

A comment on a declaration says how to use the function; a comment on a definition says how it works.

**Declarations.** Almost every function declaration gets a comment directly above it saying what it does and how to use it. Skip it only for simple, obvious functions such as trivial accessors. Private methods and functions declared in a `.cpp` are not exempt.

- **MUST** start with a verb phrase, with "This function" implied: "Opens the file", not "Open the file".
- **SHOULD** describe what, not how. The how belongs in the definition.
- **SHOULD** cover, where it applies: the inputs and outputs, with parameter names in backticks; whether the object keeps a pointer or reference argument after the call returns; whether each pointer argument may be null and what happens if it is; what happens to existing state in an output argument (appended to or overwritten); and any performance cost a caller should know about.
- **SHOULD NOT** be verbose or state the obvious. On an override, document only what is specific to the override, often nothing.
- **SHOULD** say what a constructor does with its arguments (for example, taking ownership) and what cleanup a destructor does. Skip the comment when that is trivial; most destructors need none.

```
// Returns the cached pipeline for `desc`, compiling it on first use.
// The returned reference stays valid until the cache is cleared. Compiling
// can take several milliseconds, so warm the cache at load time.
const Pipeline& PipelineCache::GetOrCompile(const PipelineDesc& desc);
```

**Definitions.** When a function does anything tricky, its definition gets a comment explaining how: the trick in use, an outline of the steps, or why this approach beat a reasonable alternative (for example, why the lock is needed for only the first half). Do not repeat the declaration comment; at most recap it in a line.

# 21

## Git and review

Commits are small, focused and written in Conventional Commits form, with no `Co-Authored-By` trailers. One commit does one thing, builds on its own, and can be reverted on its own.

### Commit format

```
<type>(<scope>): <imperative summary, under 72 characters>

<optional body: why the change was needed, what it trades off>
```

| TYPE                  | USE FOR                                         | EXAMPLE                                                               |
|-----------------------|-------------------------------------------------|-----------------------------------------------------------------------|
| <code>feat</code>     | New capability                                  | <code>feat(gpu): add timeline semaphore support</code>                |
| <code>fix</code>      | Bug fix                                         | <code>fix(cpu): sign-extend addis immediate</code>                    |
| <code>perf</code>     | Measured speedup                                | <code>perf(chunk): pool mesh buffers, 14 ms to 3 ms per remesh</code> |
| <code>refactor</code> | No behavior change                              | <code>refactor(core): split Application into Window and Loop</code>   |
| <code>test</code>     | Tests only                                      | <code>test(decoder): cover misaligned branch targets</code>           |
| <code>build</code>    | Premake, scripts, dependencies                  | <code>build: bump spdlog to 1.15</code>                               |
| <code>style</code>    | clang-format runs, renames with no logic change | <code>style: apply clang-format 19</code>                             |
| <code>docs</code>     | Docs and ADRs                                   | <code>docs(adr): choose Jolt over PhysX</code>                        |
| <code>chore</code>    | Everything else                                 | <code>chore: ignore bin-int</code>                                    |

- MUST** use the subsystem folder name as the scope (`gpu`, `cpu`, `core`, `platform`).
- MUST** keep formatting-only and rename-only changes in their own commits so real diffs stay readable.

- **MUST NOT** add `Co-Authored-By` trailers, including for AI-assisted changes.
- **SHOULD** mark breaking API changes with `! ( feat(api)!: rename prj_open to prj_session_open )` and a `BREAKING CHANGE:` footer.

### Review checklist

- ☐ Builds clean on MSVC `/W4` and Clang `-Wall -Wextra`
- ☐ clang-format and clang-tidy pass
- ☐ Names, file layout and macro prefixes follow sections 3 to 5
- ☐ Ownership is visible in every new signature; no `new` / `delete`
- ☐ Every `std::expected` is handled; every assert has a message
- ☐ No allocation added to a per-frame or per-tick path
- ☐ Threaded code states its thread affinity
- ☐ New behavior has a `.test.cpp` case; a bug fix has a regression test
- ☐ Commit messages are Conventional Commits with no co-author trailers

# A

## Appendix

---

Drop-in files for a new repository. Replace `PRJ` / `prj` / `Project` with the project's code and name.

### A..clang-format

```
Language: Cpp
Standard: Latest
BasedOnStyle: Google

# Layout (Google defaults, spelled out)
UseTab: Never
IndentWidth: 2
ContinuationIndentWidth: 4
ColumnLimit: 80
AccessModifierOffset: -1
BreakBeforeBraces: Attach
IndentCaseLabels: true
BreakConstructorInitializers: BeforeColon
ConstructorInitializerIndentWidth: 4
PackConstructorInitializers: NextLine
SpacesBeforeTrailingComments: 2
IndentPPDirectives: AfterHash
PPIndentWidth: 2

# Project choices on top of Google
NamespaceIndentation: All
FixNamespaceComments: true
InsertBraces: true
AllowShortIfStatementsOnASingleLine: Never
AllowShortLoopsOnASingleLine: false
AllowShortBlocksOnASingleLine: Empty
AllowShortFunctionsOnASingleLine: InlineOnly
DerivePointerAlignment: false
PointerAlignment: Left
ReferenceAlignment: Pointer
QualifierAlignment: Left
MaxEmptyLinesToKeep: 1

# Includes: pch, own header, project, vendor, standard library
SortIncludes: CaseSensitive
IncludeBlocks: Regroup
```

```

IncludeCategories:
- Regex: '^"[a-z]+pch\.h"$'      # precompiled header, always first
  Priority: -1
- Regex: '^<[a-z_]+>$'          # standard library
  Priority: 5
- Regex: '^<.*>$'                # vendor
  Priority: 4
- Regex: '^".*"$'                # project
  Priority: 3

```

## B. .clang-tidy

```

Checks: >
-*,
bugprone-*,
-bugprone-easily-swappable-parameters,
cppcoreguidelines-*,
-cppcoreguidelines-avoid-magic-numbers,
-cppcoreguidelines-pro-bounds-constant-array-index,
-cppcoreguidelines-pro-type-union-access,
modernize-*,
-modernize-use-trailing-return-type,
performance-*,
readability-*,
-readability-magic-numbers,
-readability-identifier-length,
misc-const-correctness,
misc-unused-*
WarningsAsErrors: 'bugprone-*,performance-*'
CheckOptions:
readability-identifier-naming.ClassCase: CamelCase
readability-identifier-naming.StructCase: CamelCase
readability-identifier-naming.EnumCase: CamelCase
readability-identifier-naming.EnumConstantCase: CamelCase
readability-identifier-naming.FunctionCase: CamelCase
readability-identifier-naming.MethodCase: CamelCase
readability-identifier-naming.NamespaceCase: CamelCase
readability-identifier-naming.VariableCase: camelBack
readability-identifier-naming.ParameterCase: camelBack
readability-identifier-naming.PrivateMemberPrefix: m_
readability-identifier-naming.PrivateMemberCase: CamelCase
readability-identifier-naming.ProtectedMemberPrefix: m_
readability-identifier-naming.ProtectedMemberCase: CamelCase
readability-identifier-naming.StaticVariablePrefix: s_
readability-identifier-naming.StaticVariableCase: CamelCase
readability-identifier-naming.GlobalVariablePrefix: g_
readability-identifier-naming.GlobalVariableCase: CamelCase
readability-identifier-naming.ConstexprVariableCase: CamelCase
readability-identifier-naming.ConstexprVariablePrefix: ''
readability-identifier-naming.MacroDefinitionCase: UPPER_CASE
readability-identifier-naming.TemplateParameterCase: CamelCase

```

## C. premake5.lua (workspace and core library)

```
-- premake5.lua
include "Dependencies.lua"

workspace "Project"
    architecture "x86_64"
    configurations { "Debug", "Release", "Dist" }
    startproject "ProjectApp"
    flags { "MultiProcessorCompile" }

    filter "system:macosx"
        architecture "ARM64"
    filter {}

outputdir = "%{cfg.buildcfg}-%{cfg.system}-%{cfg.architecture}"

group "Dependencies"
    include "Project/vendor/spdlog"
group ""

include "Project"
include "ProjectApp"
include "Sandbox"
```

```
-- Project/premake5.lua
project "Project"
    kind "StaticLib"
    language "C++"
    cppdialect "C++23"
    staticruntime "off"

    targetdir ("%{wks.location}/bin/" .. outputdir .. "%{prj.name}")
    objdir ("%{wks.location}/bin-int/" .. outputdir .. "%{prj.name}")

    pchheader "prjpch.h"
    pchsource "src/prjpch.cpp"

    files { "src/**/*.h", "src/**/*.cpp" }
    removefiles { "src/**/*.test.cpp", "src/TestMain.cpp", "src/Project/Platform/**" }
    includedirs { "src", "%{IncludeDir.spdlog}" }

    filter "system:windows"
        systemversion "latest"
        files { "src/Project/Platform/Windows/**", "src/Project/Platform/Vulkan/**" }
        buildoptions { "/permissive-", "/Zc:__cplusplus", "/Zc:preprocessor", "/utf-8" }
        warnings "Extra"

    filter "system:macosx"
        files { "src/Project/Platform/MacOS/**", "src/Project/Platform/Vulkan/**" }
        buildoptions { "-fvisibility=hidden", "-fno-omit-frame-pointer" }
```

```

        warnings "Extra"

    filter "configurations:Debug"
        defines "PRJ_DEBUG"
        runtime "Debug"
        symbols "on"

    filter "configurations:Release"
        defines "PRJ_RELEASE"
        runtime "Release"
        optimize "on"
        symbols "on"

    filter "configurations:Dist"
        defines "PRJ_DIST"
        runtime "Release"
        optimize "full"
        symbols "off"
        linktimeoptimization "on"

-- Tests: every *.test.cpp next to its source, one executable.
project "Project-Tests"
    kind "ConsoleApp"
    language "C++"
    cppdialect "C++23"
    staticruntime "off"

    targetdir ("%{wks.location}/bin/" .. outputdir .. "%{prj.name}")
    objdir ("%{wks.location}/bin-int/" .. outputdir .. "%{prj.name}")

    pchheader "prjpch.h"
    pchsource "src/prjpch.cpp"

    files { "src/prjpch.cpp", "src/**/*.test.cpp", "src/TestMain.cpp" }
    includedirs { "src", "%{IncludeDir.spdlog}", "%{IncludeDir.doctest}" }
    links { "Project" }

    filter "configurations:Debug"
        defines "PRJ_DEBUG"
        symbols "on"

```

## D. Core/Assert.h

```

#pragma once

#include "Project/Core/Base.h"
#include "Project/Core/Log.h"

#include <utility>

#if PRJ_ENABLE_ASSERTS
# define PRJ_INTERNAL_ASSERT(logger, cond, ...) \

```

```

do {
    if (!(cond)) {
        logger("Assertion failed: {} ({}:{})", PRJ_STRINGIFY(cond),
            __FILE__, __LINE__);
        logger(__VA_ARGS__);
        PRJ_DEBUGBREAK();
    }
} while (0)

# define PRJ_CORE_ASSERT(cond, ...) \
    PRJ_INTERNAL_ASSERT(PRJ_CORE_ERROR, cond, __VA_ARGS__)
# define PRJ_ASSERT(cond, ...) PRJ_INTERNAL_ASSERT(PRJ_ERROR, cond, __VA_ARGS__)
#else
# define PRJ_CORE_ASSERT(cond, ...) ((void)0)
# define PRJ_ASSERT(cond, ...) ((void)0)
#endif

// VERIFY always evaluates its condition and survives into Dist.
#define PRJ_VERIFY(cond, ...)
do {
    if (!(cond)) {
        PRJ_CORE_FATAL("Verify failed: {} ({}:{})", PRJ_STRINGIFY(cond),
            __FILE__, __LINE__);
    }
} while (0)

#define PRJ_UNREACHABLE()
do {
    PRJ_CORE_ASSERT(false, "Unreachable code reached");
    std::unreachable();
} while (0)

```

## E. Starting a new project

1. Pick the 2 or 3 letter short code and add it to the table in section 4.
2. Copy the appendix files, then rename `PRJ`, `prj` and `Project` throughout.
3. Add submodules under `vendor/` and their entries in `Dependencies.lua`.
4. Run `scripts/GenerateProjects` and confirm a clean MSVC `/W4` build of all three configurations.
5. Write the first `.test.cpp` and wire the test project into CI.
6. Write `docs/adr/0001-project-scope.md` saying what the project is and is not.