Google C++ Style Guide - Summary

1. General Principles

- Clarity: Code should be readable and understandable by others.

- Consistency: Follow the guide's conventions consistently across the codebase.

- Simplicity: Avoid complex or clever code that's hard to maintain.

- Performance: Write efficient code when performance matters but don't optimize prematurely.

2. Source File Organization

- File Naming: Use lowercase with underscores (e.g., mesh_generator.cpp).

- File Structure: One class per file if possible.

- Header Files: Use .h for declarations, .cc for definitions.

- Include Order:

1. Related header

2. C++ standard libraries

3. Other project headers

4. Third-party libraries

- Use #pragma once or include guards to prevent double inclusion.

3. Naming Conventions

- Classes/Structs: CamelCase (e.g., MeshGenerator).

- Functions: CamelCase with initial uppercase (e.g., GenerateMesh).

- Variables: snake_case (e.g., mesh_size).

- Constants: kCamelCase for global/static constants (e.g., kMaxSize).

- Enums: CamelCase with k-prefixed constants (e.g., kUnknown).

- Namespaces: snake_case (e.g., namespace robo_ex).

4. Comments & Documentation

- Use // for single-line comments, /* */ for block comments.

- Document public APIs with clear descriptions of parameters, return values, and exceptions.

- TODOs: Use // TODO(username): Description to track incomplete work.

5. C++ Language Features

- C++ Version: Use modern standards like C++14/17.

- auto: Use when the type is clear or avoids redundancy.

- Smart Pointers: Prefer std::unique_ptr and std::shared_ptr.

- nullptr: Use instead of NULL.

- constexpr: Use for compile-time constants.

- Lambdas: Use for short, localized logic.

6. Classes & Structs

- Structs: For passive data structures.

- Classes: For objects with behavior; private members go first.

- Constructors: Use =default or =delete when possible.

- Inheritance: Prefer composition over inheritance; use override.

7. Error Handling

- Assertions: Use CHECK() for invariants.

- Exceptions: Avoid; use status/error codes.

- RAII: Use resource management objects for cleanup.

8. Concurrency

- Use std::thread and std::mutex for concurrency.

- Avoid shared mutable state; prefer message passing.

- Use absl::Mutex for advanced locking.

9. Performance Best Practices

- Minimize dynamic memory allocations.

- Use emplace_back() over push_back() for efficiency.

- Avoid virtual functions in performance-critical code.

- Use move semantics (std::move) where appropriate.

10. Miscellaneous Best Practices

- Use absl libraries for Google compatibility.

- Avoid macros; use constexpr or inline functions.

- Tests: Use googletest with clear test names.

- Dependencies: Minimize dependencies and #include only when necessary.