```
// ========================================================================
// CANONICAL TECHNICAL LOCK: PORTABLE C++20 COLOR-MEANING CORE
// CONFORMANCE: GCC 11+ / CLANG 13+ (-Wall -Wextra -Wpedantic -Werror)
// ========================================================================

#include <concepts>
#include <cstdint>
#include <string_view>

// ------------------------------------------------------------------------
// 1. VALID MEANING DOMAIN & COLOR-MEANING CARRIER
// ------------------------------------------------------------------------
enum class Meaning : std::uint8_t {
    center      = 0u, // RED
    flow        = 1u, // ORANGE
    orientation = 2u, // YELLOW
    memory      = 3u, // GREEN
    motion      = 4u, // BLUE
    return_role = 5u, // INDIGO
    recognition = 6u  // VIOLET
};

inline constexpr std::uint8_t meaning_count = 7u;

[[nodiscard]] constexpr bool valid_meaning(Meaning meaning) noexcept {
    return static_cast<std::uint8_t>(meaning) < meaning_count;
}

// Finite Boolean join-semilattice over the seven valid meanings
struct SealLattice {
    std::uint8_t mask{0u};

    [[nodiscard]] constexpr bool contains(Meaning m) const noexcept {
        return (mask & (1u << static_cast<std::uint8_t>(m))) != 0u;
    }

    [[nodiscard]] constexpr SealLattice operator|(Meaning m) const noexcept {
        return SealLattice{static_cast<std::uint8_t>(mask | (1u << static_cast<std::uint8_t>(m)))};
    }

    constexpr void join_with(Meaning m) noexcept {
        mask |= static_cast<std::uint8_t>(1u << static_cast<std::uint8_t>(m));
    }
```

```
    [[nodiscard]] constexpr bool operator<=(const SealLattice& other) const noexcept {
        return (mask & ~other.mask) == 0u;
    }

    [[nodiscard]] constexpr bool operator==(const SealLattice& other) const noexcept = default;
};

// ---------------------------------------------------------------------
// 2. REFERENCE & COMPLETE STATE TYPE DEFINITIONS
// ---------------------------------------------------------------------
struct Reference {
    std::uint32_t orientation{0u};

    [[nodiscard]] constexpr bool operator==(const Reference& other) const noexcept = default;
};

struct CompleteState {
    SealLattice seal;
    Reference reference;

    [[nodiscard]] constexpr bool operator==(const CompleteState& other) const noexcept =
default;
};

[[nodiscard]] constexpr bool recognizable(const CompleteState& lhs, const CompleteState& rhs)
noexcept {
    return lhs.reference == rhs.reference;
}

// ---------------------------------------------------------------------
// 3. REFINE-TYPED REFERENCE TRANSPORT CONCEPT
// ---------------------------------------------------------------------
template<class Transport>
concept ReferenceTransport =
    requires(const Transport& transport, Reference reference) {
        { transport.admissible() } noexcept -> std::same_as<bool>;
        { transport.accepts(reference) } noexcept -> std::same_as<bool>;
        { transport.forward(reference) } noexcept -> std::same_as<Reference>;
        { transport.inverse(reference) } noexcept -> std::same_as<Reference>;
    };

// ---------------------------------------------------------------------
// 4. PORTABLE OUTCOME REPRESENTATION (NO-ALLOCATION STRUCTURAL POD)
// ---------------------------------------------------------------------
```

```
enum class OutcomeTag : std::uint8_t {
    cannot,
    failed,
    succeeded
};

enum class CannotReason : std::uint8_t {
    none,
    invalid_role,
    reference_outside_transport_domain
};

struct SucceededPayload {
    CompleteState state;
    Reference intermediate_reference;
};

struct Outcome {
    OutcomeTag tag;
    CannotReason reason;
    SucceededPayload payload;

    // Explicit structural helpers for audit visibility
    [[nodiscard]] constexpr bool is_succeeded() const noexcept { return tag ==
OutcomeTag::succeeded; }
    [[nodiscard]] constexpr bool is_cannot() const noexcept { return tag == OutcomeTag::cannot;
}
    [[nodiscard]] constexpr bool is_failed() const noexcept { return tag == OutcomeTag::failed; }
};

// ---------------------------------------------------------------------
// 5. THE GENERIC SYSTEM MACHINE ENGINE
// ---------------------------------------------------------------------
class CoreEngine {
public:
    template<ReferenceTransport Transport>
    [[nodiscard]] static constexpr Outcome apply_turn(
        const CompleteState& before,
        Meaning role,
        const Transport& transport
    ) noexcept {
        // Step A: Validate Role Boundary
        if (!valid_meaning(role)) {
            return Outcome{
```

```
            .tag = OutcomeTag::cannot,
            .reason = CannotReason::invalid_role,
            .payload = {}
        };
    }

    // Step B: Validate Reference Transport Domain Bounds
    if (!transport.accepts(before.reference)) {
        return Outcome{
            .tag = OutcomeTag::cannot,
            .reason = CannotReason::reference_outside_transport_domain,
            .payload = {}
        };
    }

    // Step C: Traverse Difference Space
    Reference intermediate = transport.forward(before.reference);

    // Step D: Evaluate Return Exactness Path
    Reference returned = transport.inverse(intermediate);
    if (!(returned == before.reference)) {
        return Outcome{
            .tag = OutcomeTag::failed,
            .reason = CannotReason::none,
            .payload = {}
        };
    }

    // Step E: Commit Structural Growth (Monotone Lift via Semilattice Join)
    CompleteState after = before;
    after.seal.join_with(role);
    after.reference = returned;

    return Outcome{
        .tag = OutcomeTag::succeeded,
        .reason = CannotReason::none,
        .payload = SucceededPayload{
            .state = after,
            .intermediate_reference = intermediate
        }
    };
  }
};
```

```
// ---------------------------------------------------------------------
// 6. √can² SYSTEM TYPE CONST EVALUATION
// ---------------------------------------------------------------------
struct Can final {};
struct CanSquared final {};
struct TurningSquare final {};

consteval CanSquared square(Can) noexcept { return {}; }
consteval TurningSquare root(CanSquared) noexcept { return {}; }

inline constexpr TurningSquare sqrt_can_squared = root(square(Can{}));

// ---------------------------------------------------------------------
// 7. COMPILE-TIME TRANSFORMS FOR EXPLICIT WITNESSES
// ---------------------------------------------------------------------
class QuarterTurnTransport {
public:
    static constexpr std::uint32_t period = 4u;

    [[nodiscard]] constexpr bool admissible() const noexcept { return true; }

    [[nodiscard]] constexpr bool accepts(Reference reference) const noexcept {
        return admissible() && reference.orientation < period;
    }

    [[nodiscard]] constexpr Reference forward(Reference reference) const noexcept {
        return Reference{.orientation = (reference.orientation + 1u) % period};
    }

    [[nodiscard]] constexpr Reference inverse(Reference reference) const noexcept {
        return Reference{.orientation = (reference.orientation + period - 1u) % period};
    }
};
static_assert(ReferenceTransport<QuarterTurnTransport>);

class TrivialIdentityTransport {
public:
    [[nodiscard]] constexpr bool admissible() const noexcept { return true; }
    [[nodiscard]] constexpr bool accepts(Reference) const noexcept { return true; }
    [[nodiscard]] constexpr Reference forward(Reference r) const noexcept { return r; }
    [[nodiscard]] constexpr Reference inverse(Reference r) const noexcept { return r; }
};
static_assert(ReferenceTransport<TrivialIdentityTransport>);
```

```
// Execute Compile-Time Audit Registers
consteval bool run_canonical_witnesses() noexcept {
    QuarterTurnTransport quarter_turn;
    TrivialIdentityTransport identity_turn;

    CompleteState state_zero{
        .seal = SealLattice{0u},
        .reference = Reference{.orientation = 0u}
    };

    // --- Witness 1: Nontrivial Turn ---
    auto outcome_1 = CoreEngine::apply_turn(state_zero, Meaning::orientation, quarter_turn);
    if (!outcome_1.is_succeeded()) return false;
    auto succ_1 = outcome_1.payload;

    if (!(succ_1.state.reference == state_zero.reference)) return false;   // Exact return
    if (!(state_zero.seal <= succ_1.state.seal)) return false;             // Monotone growth
    if (succ_1.state == state_zero) return false;                          // Complete state changed
    if (!recognizable(state_zero, succ_1.state)) return false;             // Recognition holds
    if (succ_1.intermediate_reference == state_zero.reference) return false; // Nontrivial
difference traversed

    // --- Witness 2: Trivial Transport ---
    auto outcome_2 = CoreEngine::apply_turn(state_zero, Meaning::flow, identity_turn);
    if (!outcome_2.is_succeeded()) return false;
    auto succ_2 = outcome_2.payload;

    if (!(succ_2.state.reference == state_zero.reference)) return false;   // Exact return
    if (succ_2.state == state_zero) return false;                          // Complete state grows by seal
    if (succ_2.intermediate_reference != state_zero.reference) return false; // Intermediate
reference unchanged

    // --- Witness 3: Noncanonical Reference Domain Refusal ---
    CompleteState non_canonical_state{
        .seal = SealLattice{0u},
        .reference = Reference{.orientation = 4u} // Bounds overflow for period 4
    };
    auto outcome_3 = CoreEngine::apply_turn(non_canonical_state, Meaning::orientation,
quarter_turn);
    if (!outcome_3.is_cannot()) return false;
    if (outcome_3.reason != CannotReason::reference_outside_transport_domain) return false;

    // --- Witness 4: Invalid Fabricated Meaning Refusal ---
    Meaning bad_meaning = static_cast<Meaning>(7u); // Out of 0-6 index bounds
```

```
    auto outcome_4 = CoreEngine::apply_turn(state_zero, bad_meaning, quarter_turn);
    if (!outcome_4.is_cannot()) return false;
    if (outcome_4.reason != CannotReason::invalid_role) return false;

    return true;
}

static_assert(run_canonical_witnesses());

int main() {
    return 0;
}
```