Observer Pattern

Thursday, October 30, 2025 8:05 PM

OBSERVER PATTERN LEARNING ROADMAP (C++)

Goal: Go from beginner → intermediate → project-ready mastery in C++.

one simple real-life system where *one source* of truth (a "subject") informs *many dependents* ("observers") automatically. (E.g., "A YouTube channel notifying subscribers when a new video is uploaded.")

Observer Pattern definition (in your own words):

Defines a one-to-many dependency between objects so that when one object (Subject) changes state, all its dependents (Observers) are notified and updated automatically.

PHASE 1 — Foundations (Beginner)

Focus: Core intent, roles (Subject/Observer), attach/detach/notify, pull vs push, minimal examples.

Topics

1. Problem the Observer solves (decoupled one-to-many updates)
2. Participants: Subject, ConcreteSubject, Observer, ConcreteObserver
3. attach() / detach() / notify() lifecycle
4. Push vs Pull update models (trade-offs)
5. Avoiding update loops & duplicate notifications
6. Basic sequence diagrams & event flow
7. Granularity of notifications (changed flag vs delta data)
8. Synchronous vs "queued" sync (but still single-threaded)
9. Minimal memory management choices (raw vs smart pointers)
10. Header design & compile-time dependencies

Examples (Guided)

1. **Weather Station** → Display panels auto-update
2. **Stock Ticker** → Watchlist screens update prices
3. **News Feed** → Sections subscribe to categories
4. **Chat Room (basic)** → Room broadcasts to members
5. **Button Clicks** (UI-like) → Listeners react to clicks
6. **Timer Tick** → Subscribers receive ticks
7. **Game Health Bar** → UI mirrors Player HP
8. **File Download (progress)** → Progress view subscribes
9. **Config Settings** → Listeners reload on change
10. **Logger** → Multiple sinks (console/file) subscribe

Exercises (You do)

1. Currency Rate Board
2. Temperature Logger + Alerts
3. Sports Scoreboard
4. Auction Bids feed
5. Battery Level monitors
6. Task List observers (badge counts)
7. Typing Indicator (chat)
8. Download Queue monitor
9. System Resource monitor (CPU/RAM)
10. Playlist observers (now playing / next)

PHASE 2 — Intermediate Systems

Focus: Event routing, async dispatch, thread safety, filtering, composition with other patterns.

Topics

1. **Event payload design:** push (rich event) vs pull (subject query)
2. **Observer lifetime & ownership:** weak refs, std::weak_ptr patterns

3. **Thread safety:** mutexes, lock order, reentrancy hazards
4. **Async notifications:** queues, worker threads, coalescing, debouncing
5. **Subscription filters:** topic-based, predicates, priorities
6. **Observer + Strategy / State / Command combos**
7. **Error handling:** faulty observers, timeouts, back-pressure
8. **Batch updates:** begin/end update, dirty regions
9. **Unsubscribe safety:** detach during notify, iteration guards
10. **Testing observers:** fakes/spies, deterministic dispatch

Examples (Guided)

1. **Event Bus (typed topics)** → map<string, vector<obs>> + filters
2. **Market Data Hub** → high-frequency ticks, throttling
3. **Filesystem Watcher façade** → coalesce events, debounce
4. **UI Model–View (Observer core)** → list model notifies views
5. **Chat Presence Service** → heartbeats, timeouts, retries
6. **Telemetry Pipeline** → subscriber back-pressure handling
7. **Game Event Manager** → priorities, once-only listeners
8. **Notification Center** → categories, do-not-disturb windows
9. **Sensor Fusion** → multiple sensors publish, observers aggregate
10. **Plugin System** → plugins subscribe/unsubscribe dynamically

Exercises (You do)

- Add **priority delivery** to your Event Bus
- Implement **once()** and **weak subscription** helpers
- Build a **debounced subject** wrapper (emit after quiet period)
- Implement **batched notifications** (begin/end, one notify)
- Add **per-subscriber filters & predicates**
- Make a **thread-safe dispatcher** with std::mutex + RAII guards
- Create a **record-replay** subject for testing

PHASE 3 — Applied Projects (20 mini-projects)

Focus: Realistic, multi-component systems with robust lifetime & threading.

1. **Smart Home Hub** (sensors publish; panels/automations observe)
2. **Trading Client** (ticks, PnL views, risk alerts as observers)
3. **IDE Build Monitor** (compilation events to panes)
4. **CI Dashboard** (jobs publish; subscribers: slack/email/web)
5. **Autonomous Robot Telemetry** (topic filters + rate limiting)
6. **Video Encoder Pipeline** (progress + stage events)
7. **E-commerce Inventory** (stock changes propagate to pages/carts)
8. **Document Collaboration** (model changes notify views)
9. **Navigation App** (location/traffic observers)
10. **Realtime Whiteboard** (events to cursors/shapes)
11. **Monitoring Agent** (metrics bus → multiple sinks)
12. **Build Cache Warmer** (miss/hit events drive prefetchers)
13. **Game Achievement System** (events trigger unlock logic)
14. **Email Client** (mailbox events → folders/counters)
15. **Media Player** (state + progress to UI & scrobblers)
16. **Vehicle Telemetry** (CAN events to displays/logs)
17. **Security System** (sensors → siren/log/notify)
18. **Workflow Engine** (task state updates to dashboards)
19. **Remote Debugger** (process events to views)
20. **IoT Fleet Manager** (device lifecycle & alerts)

C++ IMPLEMENTATION NOTES (across all phases)

- Prefer **interfaces with pure virtual** Observer::onEvent(...) or templated functors.
- Use **std::shared_ptr / std::weak_ptr** for observer lifetime; avoid dangling callbacks.
- For thread-safe subjects: guard **subscription list** and **notify** path with a mutex; copy the subscriber list under lock, then deliver outside the lock when possible.
- Consider **type-erased events** (std::any, variants) or **templated subjects** for zero-overhead typed channels.
- Provide **scoped subscription** helpers (RAII token that auto-detaches in destructor).
- For async: a **single producer → multi consumer** queue or a simple **dispatcher thread**.

- Beware **reentrant notifications** (observer triggers more events). Use **defer** / **queue**.
- Keep **Subject** lean; offload transformation logic to observers or dedicated processors.

QUALITY PRACTICES

Checklists

- No dangling observers; tests verify auto-detach on destruction.
- Notify cost bounded; heavy work off main thread.
- No deadlocks; lock order documented; notify outside locks when safe.
- Deterministic tests: seed data, fake time, record-replay subject.
- Back-pressure: slow observers don't block entire system.

Common Pitfalls / Anti-Patterns

- Subject owning observers strongly → cycles/leaks
- Doing business logic inside Subject → violates SRP
- Not handling detach-during-notify → iterator invalidation
- Over-notifying (chatty subject) → perf spikes
- Hidden sync across threads → races/reentrancy bugs