Inlämningsuppgift: Embedded Event Logger

Deadline: 2 februari

Bakgrund och syfte

I många inbyggda system kommer data in som händelser: en temperatur läses av, en knapp trycks, en rörelsesensor triggar. Ofta vill man kunna ta emot händelser i en kontrollerad takt, bearbeta dem i en enkel “event loop”, och logga dem för att senare kunna analysera eller felsöka systemet.

I den här uppgiften bygger du en liten firmware-lik simulator i C eller C++ (körs på datorn), med fokus på:

- tydliga datastrukturer och ADT (Abstract Data Types)
- kontrollerad minneshantering
- enkel men mätbar algoritmisk funktion (sortering och sökning)
- enkel arkitektur där du kan bygga ut stegvis

Du ska inte bygga ett stort program. Du ska bygga en stabil kärna med bra struktur.

Lärandemål (koppling till kursen)

Efter uppgiften ska du kunna visa att du kan:

- använda grundläggande datastrukturer (lista/logg och kö) på ett ändamålsenligt sätt
- resonera om tidskomplexitet för enkla algoritmer (sortering och sökning)
- skriva kod med tydliga gränssnitt (ADT) och separera “vad” från “hur”
- tillämpa minst ett designmönster (design pattern) i en enkel, motiverad form

---

Översikt: Vad du ska bygga

Du bygger ett program med tre delar:

1. **EventQueue (ringbuffer)**: tar emot inkommande händelser.
2. **EventLog (lista/logg)**: lagrar händelser som har bearbetats.
3. **EventLoop + kommandomeny**: bearbetar köade events och låter användaren interagera via terminalen.

Programmet ska kunna:

- generera och ta emot events
- bearbeta events i en loop (flytta från kö till logg)
- skriva ut loggen
- sortera loggen på timestamp
- söka efter events för en viss sensor
- välja sorteringsalgoritm via ett enkelt Strategy-upplägg (design pattern)

Krav och begränsningar

- Språk: **C eller C++**
- Programmet ska köras i terminal (ingen GUI krävs).
- Fokus ligger på **kodstruktur och korrekt minneshantering**.
- Undvik onödiga kopior av data. Spara och flytta `Event` på ett rimligt sätt.

Del 1: Kärnan (obligatorisk, krävs för Godkänt)

1. Event-modell

Definiera en `Event`-typ som minst innehåller:

- `timestamp` (int)
- `sensorId` (int)
- `type` (enum, exempelvis `TEMP`, `BUTTON`, `MOTION`)
- `value` (int)

Timestamp ska vara stigande när events skapas (till exempel genom en global räknare som ökar med 1 för varje nytt event).

---

2. ADT: EventLog (dynamisk lista/array)

Skapa en ADT som lagrar events som har bearbetats. Tanken med ADT är att resten av programmet ska använda loggen genom funktioner, utan att behöva veta hur den är implementerad internt.

Exempel på gränssnitt (du får anpassa, men behåll idén):

- `EventLog* log_create(int capacity)`
- `void log_destroy(EventLog* log)`
- `int log_size(const EventLog* log)`
- `void log_append(EventLog* log, Event e)`
- `Event log_get(const EventLog* log, int index)`
- `void log_set(EventLog* log, int index, Event e)`

Krav:

- Loggen ska vara dynamisk (kan växa vid behov eller ha en given maxkapacitet som du hanterar).
- Din kod ska hantera specialfall: tom logg, 1 element, indexkontroll på ett rimligt sätt.
- Minnet ska frigöras korrekt i `log_destroy`.

Tips: Om du väljer “växande array” kan du implementera en enkel strategi som dubblar kapaciteten när den är full.

---

3. ADT: EventQueue (ringbuffer, fast kapacitet)

Skapa en kö som representerar inkommande events. Den ska vara implementerad som en **cirkulär buffer** (ringbuffer), vilket är vanligt i embedded.

Funktioner som ska finnas:

- `Queue* queue_create(int capacity)`
- `void queue_destroy(Queue* q)`
- `bool queue_isEmpty(const Queue* q)`
- `bool queue_isFull(const Queue* q)`
- `bool queue_enqueue(Queue* q, Event e)`
- `bool queue_dequeue(Queue* q, Event* out)` (returnerar `false` om tom)

Krav:

- Kapaciteten är fast.
- Du ska inte flytta om alla element i kön vid dequeue. Ringbuffer ska göra detta effektivt.

Handhållning: En ringbuffer brukar ha:

- en array `buffer[capacity]`

- ett `head`-index (var man tar ut)
- ett `tail`-index (var man lägger in)
- en `count` (antal element)

---

4. Event loop (producer/consumer)

Implementera ett enkelt flöde där:

- “producer” skapar events (t.ex. slumpmässigt eller från en liten fördefinierad lista)
- “consumer” tar events från kön och lägger dem i loggen

Detta ska gå att trigga från kommandomenyn med exempelvis:

- `tick <n>`: kör n iterationer där programmet försöker skapa ett event och lägga i kön, och sedan försöker konsumera ett event från kön och lägga i loggen.

Du får välja exakt beteende, men det ska vara tydligt att:

- events kommer in via kön
- loggen fylls via event loop (inte genom att direkt skriva i loggen)

---

5. Sortering av loggen (minst 1 algoritm)

Du ska kunna sortera loggen efter `timestamp`.

Krav för Godkänt:

- Implementera **minst en** sorteringsalgoritm:
    - Insertion sort (rekommenderas), eller
    - Selection sort

Krav:

- den fungerar för logg med 0–1 element
- den sorterar korrekt (du ska kunna visa det)

Tips: skriv en hjälpfunktion `bool isSortedByTimestamp(const EventLog* log)` som kan användas som enkel test.

---

6. Sökning (linjär)

Implementera en enkel funktion som hittar events för en viss `sensorId` genom att gå igenom loggen linjärt.

Exempel:

- `find <sensorId>` skriver ut alla events i loggen med den `sensorId`.

---

7. Kommandomeny (terminal)

Programmet ska ha en enkel textmeny. Du kan implementera detta som en loop som läser en rad och tolkar kommandot.

Minimikrav på kommandon:

- `tick <n>`
- `print` (skriv ut loggen, eller senaste N om du vill)
- `sort <name>`
- `find <sensorId>`
- `help`
- `exit`

Du bestämmer formatet, men det ska vara lätt för en lärare att testa.

---

8. Design pattern (obligatoriskt): Strategy-light för sortering

Du ska använda ett enkelt Strategy-upplägg för att välja sorteringsalgoritm via en funktionspekare (C) eller funktionsobjekt (C++).

Tanken är:

- du har flera sorteringsfunktioner som har samma signatur
- du väljer vilken som körs baserat på textkommandot `sort ...`

Exempelidé (C):

- `typedef void (*SortFn)(EventLog* log);`
- en funktion som mappar sträng -> `SortFn`

Godkäntkrav:

- även om du bara implementerar en sort för G, ska valet ske via denna strategi. (Då visar du att du förstår mönstret och kan utöka.)

Del 2 fr.o.m. nästa sida

Del 2: Valbara utbyggnader (du väljer minst 1 för Godkänt)

Välj minst en modul nedan. Du får också föreslå en egen modul om den är relevant.

Modul A: AlarmSet (rekommenderas)

Håll reda på vilka sensorer som är i "larm".

Exempel:

- om `type == TEMP` och `value > threshold` → sensorns id ska finnas i set
- om temperaturen går tillbaka under tröskel → ta bort ur set

Kommandon:

- `alarms` listar aktiva larm
- `set-threshold <value>` (valfritt)

Implementation:

- enklast: en array med unika sensorId och en `contains`-funktion

Syftet: träna "membership test" och motivera varför man ibland vill ha en datastruktur för just detta.

Modul B: Andra sorteringsalgoritmen

Lägg till en andra sort (t.ex. den du inte valde i kärnan), och koppla den till ditt Strategy-val.

Kommandon:

- `sort insertion`
- `sort selection`

Modul C: Hash för snabb uppslagning av senaste event

Bygg en enkel struktur som gör att du snabbt kan hämta senaste event för en sensor:

- `last <sensorId>`

Implementation kan vara enkel:

- en array om sensorId är begränsade, eller
- en egen liten hashtabell (för den som vill)

Modul D: Observer-light (bonus)

När ett event processas i event loopen, notify:a två “lyssnare”:

- `Logger` som lägger event i loggen
- `AlarmManager` som uppdaterar AlarmSet

Syftet: separera ansvar och visa ett designmönster som passar eventdrivna system.

Rapport (kort och tydlig)

Skriv en kort rapport (cirka 1 sida) som innehåller:

1. **Översikt över din lösning**
    - vilka moduler/ADT:er du har
    - hur event-flödet ser ut (Queue → Loop → Log)
2. **Design pattern**
    - vad Strategy innebär i din kod och varför det passar här
3. **Tidskomplexitet (grovt)**
    - din sortering (best/worst i stora drag)

    - din sökning (linjär)

4. **Minneshantering**
    - var allokerar du minne och var frigör du det
    - nämn kort hur du har tänkt undvika läckor

Inlämning

Lämna in:

- källkod eller länk till repo (med tydlig mappstruktur)
- README med:
    - hur man bygger och kör
    - exempel på ett par testkommandon (en “körning” som visar att det fungerar)
- rapporten

Rekommenderad struktur:

- `src/` (implementation)
- `include/` (headers)
- `main.c` / `main.cpp`
- `README.md`
- `report.pdf` eller `report.md`

Bedömning

Godkänt (G)

För Godkänt ska du ha:

- Kärnan klar: `Event`, `EventQueue`, `EventLog`, event loop, meny
- Minst 1 sortering kopplad via Strategy-light
- Linjär sökning (`find`)
- Minst 1 valfri modul (A, B, C eller D)
- Rimlig kodstruktur med ADT i separata filer
- Korrekt minneshantering vid normal körning

Väl godkänt (VG)

För Väl godkänt ska du ha:

- Allt i G
- Minst 2 sorteringsalgoritmer, valbara via Strategy
- Minst 2 utbyggnadsmoduler (exempel: AlarmSet + Hash-last, eller AlarmSet + Observer)
- Snygg och tydlig modulindelning, välmotiverade val
- Rapport som visar bra resonemang om tradeoffs och komplexitet

---

Tips för att lyckas

- Börja med att få programmet att kompilera och köra med en minimal meny.
- Bygg ADT:erna tidigt och håll dem små och tydliga.

- Skriv små testfall: skapa 5–10 events, kontrollera att `print`, `find` och `sort` fungerar.

- Om du fastnar: förenkla. Ett litet, korrekt system slår ett stort som inte fungerar.