Trabajo Práctico 2

I102 - Paradigmas de Programación
Lautaro Valentín Caminoa y Francisco Krinisky

Universidad de San Andrés - 2025

Resumen

Este informe presenta la solución al Trabajo Práctico 2 de la materia Paradigmas de Programación, compuesto por tres ejercicios independientes. Cada uno aborda distintos conceptos de programación en C++: estructuras de datos con hashing, multithreading y sincronización entre threads. El Ejercicio 1 implementa una Pokédex con *unordered_map* y serialización binaria; el Ejercicio 2 simula drones que deben acceder a zonas críticas mediante *mutex* y *lock*; y el Ejercicio 3 desarrolla un sistema tipo productor-consumidor con *condition_variable*.

1. Introducción

El objetivo del Trabajo Práctico 2 es afianzar el uso de programación orientada a objetos, STL containers y técnicas básicas de concurrencia. Cada ejercicio propone un problema concreto cuya solución implica un diseño modular (organizados en distintos directorios), el uso apropiado de estructuras de datos y la correcta sincronización entre threads para evitar race conditions o bloqueos.

2. Metodología

La implementación se realizó de manera modular, utilizando archivos *.hpp* y *.cpp* separados para cada componente, junto con Makefiles individuales para compilar los ejecutables. Se utilizó *g++* con los flags *-std=c++17 -Wall -Wextra -Wpedantic*. La correcta funcionalidad de cada sistema fue verificada mediante casos de prueba en *main.cpp*, incluyendo validación de salidas esperadas y detección de errores comunes.

3. Descripción de la Implementación

3.1 Ejercicio 1: Pokédex (OOP + Containers + Hashing)

3.1.1 Objetivo

Diseñar una estructura que permita registrar, consultar, guardar y restaurar información de múltiples Pokémon. Se hace uso de técnicas de hashing para eficiencia en acceso, y de serialización binaria para persistencia.

3.1.2 Implementación

El sistema se compone de tres clases principales:

- *Pokemon*: almacena nombre *(std::string*) y experiencia (*int*). Implementa:
  - Constructores por defecto y parametrizados.
  - Getters.
  - Operador ==.
  - Métodos *serializar(std::ofstream&)* y *deserializar(std::ifstream&)*.
- *PokemonInfo*: contiene:
  - Tipo y descripción *(std::string*).
  - Ataques (*std::map<std::string, int>*).
  - Experiencia necesaria para subir de nivel (*std::array<int, 3>*).
  - Métodos *serializar()* y *deserializar()*.
- *Pokedex*: núcleo del sistema.
  - Usa *std::unordered_map<Pokemon, PokemonInfo>* como estructura interna.
  - Sobrecarga *std::hash<Pokemon>* para clave personalizada.
  - Incluye *agregar*, *mostrar*, *mostrarTodos*, *guardarEnArchivo()* y *cargarDesdeArchivo()*.
  - Tiene un constructor sobrecargado que permite inicializar desde archivo automáticamente.

3.1.3 Pruebas

- Se agregaron *Squirtle*, *Bulbasaur* y *Charmander* con ataques, tipo, experiencia.
- Se ejecutó *mostrarTodos()*.
- Se validó *mostrar()* con Pokémon existentes y no existentes.
- Se creó una Pokedex en archivo binario, se serializó, y luego se leyó con una nueva instancia para verificar persistencia.

3.1.4 Compilación

El ejercicio cuenta con su propio *Makefile* que incluye los flags correspondientes para los warnings. Para correr el ejercicio, pararse en el directorio correspondiente y utilizar el comando *make* en la terminal para compilarlo (lo cual genera el ejecutable *Ej1*), seguido del comando *./Ej1* para ejecutarlo.

3.2 Ejercicio 2: Drones (Multithreading + Mutex)

3.2.1 Objetivo

Modelar el despegue de 5 drones en círculo. Cada uno debe adquirir sus dos zonas adyacentes antes de despegar. La protección de zonas se realiza mediante mutex y locking múltiple.

3.2.2 Implementación

- *Hangar*: contiene 5 mutex *(std::mutex zonas[5]*) representando zonas.
- *Drone*: recibe su *id*, puntero a zonas y al mutex de salida. En *despegar()*:
    - Determina sus zonas *(der = id*, *izq = (id + 1) % 5*).
    - Usa *std::lock* para evitar deadlocks.
    - Protege cada zona con *lock_guard* + *adopt_lock*.
    - Imprime mensajes ordenados con *mutex_out*.
    - Despegue simulado con *sleep_for(5s)*.
- En *main()*:
    - Se crean los 5 drones y sus threads.
    - Se ejecuta cada *despegar()* en paralelo y se sincroniza con *join()*.

3.2.3 Pruebas

- Se verificó que no hubiera interbloqueos.
- Se validó que los mensajes salgan ordenados y sin superposiciones.
- Se inspeccionó que cada dron tenga acceso exclusivo a sus zonas.

3.2.4 Compilación

El ejercicio cuenta con su propio *Makefile* que incluye los flags correspondientes para los warnings. Para correr el ejercicio, pararse en el directorio correspondiente y utilizar el comando *make* en la terminal para compilarlo (lo cual genera el ejecutable *Ej2*), seguido del comando *./Ej2* para ejecutarlo.

3.3 Ejercicio 3: Sistema de Monitoreo (Multithreading + Condition Variable)

3.3.1 Objetivo

Simular sensores que generan tareas y robots que las procesan, usando una cola compartida sincronizada. El diseño replica el patrón productor-consumidor.

3.3.2 Implementación

Archivo único *Ej3.cpp*. Componentes:

- *struct Tarea*: contiene *idSensor*, *idTarea* y *descripcionTarea*.
- *std::queue<Tarea> colaTareas*: cola compartida.
- *std::mutex mtxCola*: acceso a la cola.
- *std::condition_variable cvTareas*: para bloqueo de consumidores.
- *std::mutex mutex_out*: evita solapamiento en consola.
- *bool sensoresFinalizados*: indica si ya no habrá nuevas tareas.

Funciones:

- *sensor()*: genera tareas con retardo de 175ms. Cada tarea se encola con *lock_guard*. El *notify_one()* se hace dentro del mismo *mutex_out* que imprime.
- *robot()*: espera tareas con *cvTareas.wait()*. Procesa con *sleep_for(250ms)*. Sale si la cola está vacía y los sensores terminaron.
- *main()*: lanza 3 sensores y 3 robots. Al finalizar los sensores, setea *sensoresFinalizados = true* con *mutex*, y hace *notify_all()*.

3.3.3 Pruebas

- Se verificó la correcta sincronización y salida ordenada.
- Se testeó que cada tarea sea procesada solo una vez.
- Se validó la finalización de robots al terminar los sensores.

3.3.4 Compilación

El ejercicio cuenta con su propio *Makefile* que incluye los flags correspondientes para los warnings. Para correr el ejercicio, pararse en el directorio correspondiente y utilizar el comando *make* en la terminal para compilarlo (lo cual genera el ejecutable *Ej3*), seguido del comando *./Ej3* para ejecutarlo.

4. Warnings del Compilador

La compilación de todos los ejercicios se realizó con los flags *-Wall -Wextra -Wpedantic* para detectar posibles errores o malas prácticas. Todos los warnings fueron corregidos, resultando en un código limpio y sin advertencias al compilar.

5. Conclusiones

El TP2 fue completado en su totalidad, incluyendo la parte opcional del Ejercicio 1 mediante serialización binaria. Se abordaron distintas técnicas de C++, como containers, hashing, exclusión mutua, sincronización, y programación concurrente. El diseño modular y los distintos tests permitieron validar la robustez y claridad de las soluciones presentadas.