Trabajo Práctico 2

I102 – Paradigmas de Programación

Autores: Luca Fassio y Bautista García

Grupo 1

Profesor: Mariano Scaramal

Universidad de San Andrés (UdeSA)

Año: 2025

Introducción

Este informe tiene como objetivo documentar la resolución del Trabajo Práctico 1 de la materia Paradigmas de Programación. A lo largo de los ejercicios se abordaron conceptos avanzados del lenguaje C++, tales como serialización binaria, manejo de punteros inteligentes, templates, *type traits*, *concepts*, programación genérica y programación concurrente con `thread` y `mutex`.

Cada ejercicio fue implementado en un proyecto independiente con su propio entorno de compilación utilizando **CMake**. Para compilar y correr cada programa, se debe ejecutar:

1. `cd ejX/build`
2. `cmake ..`
3. `make`
4. `./bin/main`

donde `ejX` representa el directorio del ejercicio correspondiente (por ejemplo, `ej1`, `ej2`, `ej3`).

A continuación se detalla la resolución de cada ejercicio junto con las decisiones de diseño, complicaciones encontradas y aspectos destacados de cada implementación.

Ejercicio 1 – Pokedex digital en C++

Main Classes

En este ejercicio se implementó una versión simplificada de una **Pokedex digital**. El objetivo principal era modelar y almacenar la información relevante de distintos Pokemones, permitiendo acceder y mostrar sus datos de manera eficiente.

Se definieron tres clases principales para organizar la información:

- **Pokemon**: representa un Pokemon específico y contiene atributos como el nombre y experiencia. Se sobrecargaron los operadores necesarios para que pueda ser utilizado como clave en un `unordered_map` y se implementó una función de hash que los ubica en base al nombre.
- **PokemonInfo**: almacena la información complementaria de cada Pokemon, incluyendo el tipo (agua, fuego, planta, etc.), una breve descripción, los ataques disponibles (con su puntaje de poder) y la experiencia necesaria para alcanzar sus tres niveles. Para los ataques y la experiencia, se optó por usar dos vectores, ya que consideramos que son estructuras de datos adecuadas que nos sirven para facilitar tanto la consulta como la impresión por pantalla.
- **Pokedex**: actúa como contenedor principal, utilizamos un `unordered_map<Pokemon, PokemonInfo, Hash>`, lo que permite búsquedas rápidas de la información a partir de una instancia de Pokemon. Se implementó un functor de hash personalizado, como se mencionó en la clase Pokemon, considerando únicamente el nombre para calcular el hash.

Serializing y Deserializing

Para cumplir con los puntos extra del ejercicio, se implementó la funcionalidad de guardar y cargar los datos de la Pokedex utilizando archivos binarios. El objetivo principal fue permitir que la información registrada por el usuario permanezca disponible entre distintas ejecuciones del programa.

La serialización de datos se realiza a través del método `saveToFile` de la clase `Pokedex`. Al invocar este método, se almacena en un archivo binario la cantidad total de Pokemones registrados y, para cada uno, se serializan los datos esenciales utilizando el método `serialize` de la clase `Pokemon`. Es importante destacar que en este proceso **no se serializa la información contenida en la clase** `PokemonInfo`, ya que cada Pokemon tiene una única instancia asociada y dicha información se reconstruye automáticamente al cargar los datos.

Para restaurar la Pokedex desde un archivo binario, se implementó el método `loadFromFile`. Durante este proceso, se deserializan los datos de los Pokemones y se reconstruye la Pokedex. Dado que la carga de grandes cantidades de Pokemones al principio era una tarea costosa (llegando a demorar hasta 25 segundos), se optó por utilizar **multithreading** para acelerar la operación. En concreto, se divide el conjunto de Pokemones deserializados en bloques, y cada bloque es procesado por un hilo independiente, encargándose de agregar su segmento

correspondiente a la Pokedex. Esto permite aprovechar los recursos del sistema y lograr una carga más eficiente cuando la cantidad de datos es significativa.

Luego de aplicar esto nos percatamos de que no cambiaba mucho, ya que el problema de la mala optimización no se resolvía al paralelizar el proceso, sino que era generado por abrir y cerrar repetidas veces los archivos CSV que utilizamos para extraer la información de los 802 Pokemones más `pokeAgusRoca`. Para solucionar esto, al iniciar el programa se carga toda la información de los CSV en dos `unordered_maps`, lo cual aceleró el proceso de carga (siempre comparando con la Pokedex "All Unlocked" que tiene a todos los Pokemones desbloqueados) hasta hacerlo prácticamente instantáneo, dependiendo de la computadora utilizada.

En definitiva, el sistema de persistencia implementado logra conservar y restaurar la información esencial de cada Pokemon de manera eficiente, evitando la redundancia de datos y aprovechando tanto la reconstrucción dinámica de `PokemonInfo` como el uso inteligente de múltiples hilos durante la carga masiva.

Visualización e impresión de Pokemones

Para la visualización de la Pokedex y la impresión de los datos de los Pokemones, se desarrolló un sistema que combina tanto la presentación textual como la inclusión de imágenes en la terminal. El objetivo principal fue lograr una experiencia más interactiva y visual para el usuario, emulando un intento de la estética de una Pokedex "real".

Cada página de la Pokedex muestra hasta quince Pokemones, distribuidos en filas de tres, donde para cada uno se imprime la imagen y la información correspondiente. La lógica para construir este layout se implementó en los métodos `show` y `printThreePokes` de la clase `Pokedex`, apoyándose en varias funciones auxiliares presentes en los archivos `pokedexHUD.cpp` y `utils.cpp`.

La inclusión de imágenes de los Pokemones se realiza utilizando una **librería externa** [2] para la carga de imágenes en formato PNG, específicamente `stb_image.h` (**disponible acá** [2]). Esta librería permite abrir, procesar y transformar las imágenes en tiempo de ejecución a un `unsigned char*` con la información de RGB o RGBA de cada pixel, para luego pasarlas por un proceso de resize y poder mostrarlas directamente en la terminal mediante caracteres ANSI de distintos colores.

Las imágenes de los Pokemones se encuentran almacenadas en la carpeta `assets/imgs/` y se seleccionan en función del número de Pokedex de cada Pokemon. El archivo CSV utilizado fue obtenido del TP final de Pensamiento Computacional (otoño 2024), con la única modificación de agregar descripciones extraídas de la página [3]. Además, se unificaron los Pokemones Meowstic-F y Meowstic-M en un único "Meowstic", ya que representan el mismo Pokemon en su versión femenina y masculina, comparten número de Pokedex y `PokemonInfo`, y esto generaba problemas a la hora de almacenarlos.

En definitiva, la visualización implementada combina técnicas de procesamiento de imágenes, formateo de texto y control del layout en consola, utilizando principalmente funciones propias pero apoyándose en una librería externa para lograr una experiencia de usuario completa y fiel a la temática de la Pokedex.

BONUS:

En la Pokedex "All Unlocked", ademas de tener la posibilidad de buscar cualquier Pokémon disponible en el TP Final de PC mencionado (hasta la generación 7), pueden además buscar a los Pokémones Legendarios UdeSA Exclusive Edition: Agus Roca y Juani.

Igualmente, en cualquier Pokedex (ya sea una nueva o una de las que utilizamos para probar cosas durante el TP que dejamos en la carpeta) se puede agregar y desbloquear estos Pokémones iniciando con 0 de xp como cualquier otro que desbloquee el usuario.

Ejercicio 2 – Simulación de despegue de drones con sincronización por zonas compartidas

Este ejercicio consistía en simular el proceso de despegue de **cinco drones** dispuestos en una ronda. Cada dron requería **dos zonas adyacentes** para poder despegar: la propia y la siguiente (tomando el arreglo como circular, es decir, el dron 4 utiliza la zona 4 y la zona 0). Una vez que ambas zonas estuvieran libres, el dron debía "ocuparlas", realizar el procedimiento de despegue (simulado con una espera de 5 segundos), alcanzar los 10 metros de altura y luego liberar las zonas. El objetivo principal era minimizar el tiempo total de despegue, evitando que los drones se bloqueen entre sí innecesariamente.

Para representar las **zonas de despegue** se utilizó un arreglo de `mutex`, uno por cada zona (cinco en total). La lógica del despegue se implementó dentro de una función ejecutada por cada hilo (`thread`), en donde se identificaban las dos zonas necesarias: `left = id` y `right = (id + 1) % 5`. Donde `id` es el numero del dron que estamos queriendo despegar.

Inicialmente, se intentó bloquear los mutex por separado, pero en ciertas ejecuciones esto generaba **deadlocks** cuando varios drones esperaban mutuamente por zonas que otros ya habían bloqueado. Para resolver este problema, se pasó a utilizar `lock` en conjunto, lo que permite adquirir ambos mutex de forma atómica, eliminando así la posibilidad de deadlocks.

Con esta solución nos evitamos condiciones de carrera, eliminamos el riesgo de deadlocks y **minimizamos el tiempo de ejecución total** al permitir que varios drones despeguen simultáneamente si no comparten zonas.

Además, se estructuró el programa para que sea **escalable**: encapsulando la lógica en una clase que mantiene tanto el arreglo de zonas (`mutex zonas[5]`) como el vector de hilos (`vector<thread> drones`). Esto facilita extender el sistema a una cantidad mayor de drones o zonas sin modificar el funcionamiento interno del programa.

Ejercicio 3 – Simulación de procesamiento de tareas concurrentes

En este ejercicio se buscó modelar un sistema con múltiples sensores y robots, donde los primeros generan tareas y los segundos las procesan. Para ello, se usaron herramientas como `thread`, `mutex` y `condition_variable`, con el objetivo de evitar condiciones de carrera y lograr una ejecución concurrente correcta.

Los sensores fueron implementados como hilos que, al ejecutarse, generan una cantidad aleatoria de tareas (entre 1 y 5). Cada tarea contiene un id del sensor que la creó, un id único global (ó `uuid`) y una breve descripción. Estas tareas se almacenan (luego de 175 milisegundos) en una cola compartida entre todos los hilos, protegida por un mutex. Al insertar una tarea, el sensor invoca `notify_one()` para que un robot en espera pueda reanudar su ejecución. Una vez que cada sensor termina de insertar todas sus tareas, incrementa una variable global `sensorsFinished`, que indica cuántos sensores han finalizado su trabajo. Dado que esta variable también es compartida, su modificación está protegida.

Por otro lado, los robots también son hilos independientes que trabajan procesando las tareas disponibles. Cada robot comienza con una espera de 250 milisegundos, simulando el tiempo de trabajo requerido. Luego adquiere un `unique_lock` sobre el mutex y se bloquea mediante un `cv.wait()` que permanece en pausa hasta que haya tareas disponibles o todos los sensores hayan terminado. Esta espera se acompaña de una condición que evita los denominados *spurious wakeups*. En caso de que no queden tareas y todos los sensores hayan finalizado, el hilo del robot se detiene. Caso contrario, el robot toma la tarea más antigua de la cola, la elimina, y libera el mutex para permitir que otros hilos puedan acceder al recurso compartido.

El uso de `unique_lock` fue necesario para poder integrar el mecanismo de espera de la `condition_variable`. La lógica de sincronización, junto con la verificación del estado global del sistema mediante `sensorsFinished`, permitió diseñar una solución robusta al problema del productor-consumidor. Toda la estructura fue pensada para escalar sin mayores cambios, de modo que se pueden aumentar tanto la cantidad de sensores como de robots sin modificar la lógica general del sistema.

References

[1] Mike Shah. *Introduction to Concurrency in Cpp*. https://youtu.be/hXKtYRleQd8?si=sePO8DyLzCXCmVuZ. Playlist de YouTube. 2022.

[2] Sean Barrett. *stb - single-file public domain libraries for C/C++*. https://github.com/nothings/stb. Accedido el 6 de junio de 2025. 2024.

[3] PokeAPI Team. *PokeAPI: The RESTful Pokémon API*. https://pokeapi.co/. Accedido el 6 de junio de 2025. 2024.