Universidad de
SanAndrés

Trabajo Práctico 2 – Paradigmas de Programación

Profesor: Mariano Scaramal

Autores: Franco Needleman y Lautaro Lo Presti

Universidad de San Andrés (UdeSA)

Año: 2025

Introducción

En este Segundo Trabajo Práctico de la materia **Paradigmas de Programación**, abordamos tres problemas independientes utilizando los estándares **C++20** y **C++17**, aplicando estructuras de datos de la STL, serialización binaria y técnicas de programación concurrente. Cada ejercicio fue organizado en su propia carpeta y compilado mediante **CMake**. Para compilar, creamos un directorio build dentro de cada ejercicio y, desde allí, ejecutamos:

```
cmake ..

make

./ejercicioX
```

donde X representa el número del ejercicio. Este esquema nos permitió una compilación limpia, modular y fácilmente reproducible en distintos entornos.

En los archivos CMakeLists.txt se especificó el estándar de C++ correspondiente, se añadió la carpeta headers al include path, y mediante un glob se recopilaron automáticamente todos los archivos fuente de sources/*.cpp junto con el mainX.cpp para generar el ejecutable ejercicioX. Se activaron las opciones de compilación estrictas -Wall, -Wextra y -Wpedantic para detectar advertencias y mantener un alto estándar de calidad. No se registraron advertencias durante la compilación, lo que indica que el código fue implementado correctamente y con buenas prácticas.

Ejercicio 1 – Pokédex digital en C++

En este primer ejercicio diseñamos una **Pokédex** digital capaz de almacenar, buscar, mostrar y guardar en disco información sobre distintas especies de Pokémon.

La clase Pokemon contiene dos atributos: el nombre del ejemplar y su experiencia actual. Se sobrecargó el operador == y se implementó un functor de hash, ambos basados únicamente en el nombre del Pokémon. Esto permite usar instancias de Pokemon como claves en un std::unordered_map, lo que garantiza búsquedas rápidas en promedio O(1) y evita inconsistencias si == y hash usan distintos criterios.

La clase PokemonInfo agrupa el tipo elemental (agua, fuego, etc.), una descripción, un std::map que asocia ataques con su daño, y un std::vector con la experiencia requerida para alcanzar los tres niveles posibles. Elegimos map para los ataques por su ordenamiento automático y eficiencia en búsquedas puntuales, y vector para las experiencias porque permite acceso directo por índice. Ambos contenedores son fáciles de usar, extensibles y no requieren sobrecargas adicionales para su funcionamiento.

La clase Pokedex funciona como contenedor principal, usando un unordered_map<Pokemon, PokemonInfo, PokemonHash>. Al invocar el método agregar, se inserta una nueva entrada y luego se serializa todo el contenido en el archivo pokedex.dat en modo binario y truncado. Primero se escribe la cantidad total de Pokemones, y luego cada uno con sus datos: nombre, tipo, descripción, ataques y experiencia por nivel. Antes de cada texto, se guarda su longitud para facilitar la deserialización.

Para reconstruir los datos, se implementó un constructor que recibe un ifstream abierto y llama al método deserializar, cargando automáticamente toda la información en memoria.

La visualización se realiza con dos métodos: mostrar(const Pokemon&), que imprime nombre, experiencia, tipo, descripción, ataques y calcula dinámicamente el

nivel alcanzado y la experiencia faltante; y mostrarTodos(), que recorre todo el mapa reutilizando esa lógica.

Durante las pruebas se agregaron tres Pokemones: **Squirtle (100 exp)**, **Bulbasaur (270 exp)** y **Charmander (633 exp)**. Llamar a mostrar("Squirtle", 870) confirmó que el programa encuentra correctamente a los Pokemones por nombre, ignorando la experiencia del objeto de búsqueda. En cambio, mostrar("Pikachu", 500) mostró el mensaje "¡Pokemon desconocido!", validando el manejo de búsquedas fallidas. Al reiniciar el programa y cargar desde el archivo binario, los datos se restauraron sin pérdida de información.

Ejercicio 2 – Simulación de despegue de drones con sincronización

Este ejercicio simula el **despegue coordinado de cinco drones** dispuestos en círculo. Cada dron debe asegurarse de que las dos zonas adyacentes a su posición estén libres antes de despegar, lo que plantea un problema clásico de sincronización con riesgo de **deadlocks**.

Implementamos una clase Drones, que contiene un arreglo de cinco std::mutex llamado areas (una por zona de interferencia) y un mutex adicional terminal para proteger las impresiones en consola.

El método simularVuelos() lanza cinco hilos mediante std::jthread, cada uno invocando el método despegar(int drone) con identificadores aleatorios (sin repetir). Usamos jthread porque libera automáticamente el recurso al salir de scope, eliminando la necesidad de llamar a join() manualmente y evitando errores comunes con hilos.

Dentro de despegar, primero se bloquea terminal para imprimir que el dron está esperando. Luego, se llama a std::lock(areas[id], areas[(id+1)%5]), lo que bloquea las dos zonas adyacentes de forma **atómica**, previniendo interbloqueos. Si los mutex se tomaran por separado, podrían producirse situaciones de espera circular. Con std::lock, los mutex se adquieren juntos o no se adquieren, garantizando seguridad.

Después de imprimir que está despegando, el hilo duerme 5 segundos para simular el ascenso y luego libera manualmente ambos mutex. terminal se utiliza exclusivamente para que las salidas a consola no se solapen entre hilos concurrentes, asegurando mensajes ordenados y legibles.

Cada ejecución presenta un orden distinto de impresión debido a que la ejecución simultánea de los threads no garantiza un orden específico, sin embargo, en todas se verifica que los drones esperan correctamente, despegan sin interferencias y alcanzan la altura sin bloqueos. Esto demuestra un uso correcto de sincronización concurrente junto a un buen uso de std::lock() y mutex.

Ejercicio 3 – Sistema de monitoreo y procesamiento de robots autónomos

En este ejercicio implementamos un **patrón productor-consumidor**, donde sensores generan tareas y robots las procesan en paralelo.

Se define un struct Tarea con tres campos: el ID del sensor, un ID de tarea único y una descripción. Las tareas se almacenan en una std::queue<Tarea> protegida por mutexCola, y la generación de IDs se sincroniza mediante mutexTareas para asegurar exclusión mutua al incrementar numeroTareas.

Un tercer mutex terminal se emplea para proteger las salidas por consola. También usamos una std::condition_variable cvCola para permitir que los robots esperen eficientemente hasta que haya tareas disponibles, en lugar de estar verificando constantemente la cola (lo que ahorra CPU).

Cada uno de los tres sensores (productores) ejecuta una función que se repite cinco veces. En cada iteración, el sensor duerme 175 ms, asigna un nuevo ID (protegido por mutexTareas), encola la tarea (lock_guard de mutexCola), notifica a los robots con cvCola.notify_all(), e imprime lo que hizo.

Simultáneamente, tres hilos de robots (consumidores) esperan en cvCola hasta que haya tareas o hasta que todos los sensores hayan terminado. Para ello, usamos una bandera sensoresFinalizados. Esta bandera indica a los robots cuándo ya no se generarán más tareas. Si no se usara, podrían quedar esperando indefinidamente.

Cuando los sensores finalizan, el main activa la bandera, hace un notify_all() final para despertar a los robots, y luego espera a que estos terminen. Los robots procesan cada tarea extrayéndola de la cola, imprimiendo la acción, simulando un procesamiento de 250 ms, y repitiendo el ciclo hasta que la cola queda vacía y no habrá más tareas.

El programa concluye mostrando el mensaje “Todos los sensores y robots han finalizado.”, validando que toda la lógica de sincronización, protección de recursos compartidos y comunicación entre hilos se implementó de forma correcta, sin condiciones de carrera ni bloqueos indeseados.