Universidad de San Andrés

Trabajo Práctico 2

I102 – Paradigmas de Programación – Grupo:1

Franco Sandri, Martín De Hieronymis

Universidad de San Andrés

1er Semestre 2025

En este informe se presenta el desarrollo y la solución de los ejercicios planteados en el Trabajo Práctico N.º 2. El objetivo principal es aplicar conceptos de programación orientada a objetos y programación concurrente en C + +, resolviendo problemas prácticos.

1. Pokédex.

En primer lugar, se diseñó un sistema de Pokédex, utilizando múltiples clases para una estructura clara, mantenible y extensible. El sistema permite agregar Pokémon, visualizarlos y almacenarlos de forma persistente en un archivo binario.

Estructura del ejercicio

El ejercicio está dividido en tres carpetas:

- headers: contiene los archivos headers (.hpp) para las clases Pokedex, Pokemon, PokemonInfo.
- src: contiene los archivos de implementación (.cpp) que corresponden a los headers.
- main: contiene el archivo main, donde se prueba el correcto funcionamiento de las clases.

Clases

Pokemon: esta clase representa un pokémon con dos atributos.

- nombre: string que representa el nombre del pokémon.
- experiencia: int que representa su experiencia acumulada.

En esta clase se implementan operadores (==, <), para el uso de la librería map y las funciones getExperiencia() y getNombre() que devuelven el los atributos del pokémon respectivamente. Por otro lado, se realizan las validaciones de que el nombre no sea un string vacío y que la experiencia no sea negativa.

PokemonInfo: esta clase amplía la descripción de cada uno de los pokemons con cuatro atributos adicionales.

- Pokemon: pokémon del cual pertenece la información.
- tipo: string que define que tipo es el pokémon.

- descripcion: string con una descripción del pokémon.
- ataquesDisponiblesPorNivel: map<string, int> con el nombre del ataque y su daño.
- experienciaProximoNivel: array de enteros con la experiencia requerida para los siguientes niveles.

En esta clase se implementan las funciones get y validaciones para cada uno de los atributos.

PokemonHash: esta clase se crea para poder utilizar objetos de la clase Pokemon como clave en estructuras de datos no ordenadas como unordered_map. Esta clase utiliza el hash estándar de strings(hash<string>) aplicado al nombre del Pokémon, ya que en esta implementación, el nombre es el identificador único de un Pokémon. Este hash personalizado es esencial para garantizar el correcto funcionamiento de la estructura unordered_map<Pokemon, PokemonInfo, PokemonHash> usada internamente en la clase Pokedex.

Pokedex:

Esta es la que administra todos los pokémons, con dos atributos.

- pokedex: unordered_map<Pokemon, PokemonInfo, PokemonHash> es el contenedor principal. Almacena pares Pokemon (como clave) y PokemonInfo (como valor), utilizando un unordered_map con un hash personalizado (PokemonHash).
- archivo: string que es el nombre del archivo binario donde se guarda y carga la información de la pokédex.

Esta clase está encargada de almacenar y gestionar información sobre múltiples Pokémons. Actúa como una base de datos que puede agregar pokémons nuevos (agregarPokemon()), mostrar información de un pokémon específico (mostrar(const Pokemon&)) y mostrar todos los pokémons registrados (mostrarTodos()). Y dentro tiene implementada las funciones cargarDesdeArchivo() que lee el contenido de un archivo binario y reconstruye la pokédex y guardarEnArchivo() que guarda todo el contenido de la pokédex al archivo binario.

Main

El archivo main.cpp actúa como una prueba del correcto funcionamiento del sistema, demostrando el uso práctico de la clase Pokedex. En primer lugar, se crea una instancia de Pokedex asociada al archivo binario "pokedex.dat". Luego, se agregan tres Pokémon (Squirtle, Bulbasaur y Charmander) con sus respectivas experiencias, descripciones y ataques. Posteriormente, el sistema intenta mostrar la información de un Pokémon no registrado (Pikachu), lo que permite verificar el manejo de búsquedas fallidas. La función mostrarTodos() se utiliza para listar en consola todos los Pokémon almacenados, validando la correcta inserción y visualización. Finalmente, todo se encuentra dentro de una estructura de manejo de excepciones para asegurar que cualquier error durante la ejecución (como datos

inválidos o fallos en el archivo) sea capturado de forma segura, evitando caídas inesperadas del programa

Implementación

A lo largo de todo el ejercicio se utilizan estructuras como unordered_map y array, junto con serialización binaria, para optimizar el rendimiento y la persistencia de los datos. Además, se incorporan mensajes por consola para facilitar la trazabilidad y depuración del programa.

El uso de unordered_map y array en la implementación de la clase Pokedex permite una representación de datos eficiente y compacta. El unordered_map facilita búsquedas rápidas de información asociada a cada Pokémon, utilizando como clave su nombre y experiencia encapsulados en la clase Pokemon, mientras que el array garantiza un almacenamiento de tamaño fijo y acceso constante para los niveles de experiencia requeridos para la evolución. Además, la serialización binaria empleada en las funciones de carga y guardado permite almacenar los datos de forma compacta y eficiente en disco, asegurando compatibilidad entre ejecuciones. A esto se suma la inclusión de mensajes informativos por consola, los cuales permiten un seguimiento detallado del flujo de ejecución del programa, facilitando la depuración, el control de errores y la comprensión del comportamiento interno del sistema durante su uso o desarrollo.

2. Control de Aeronave en Hangar Automatizado.

En el siguiente apartado se describe la implementación de un sistema de control de despegue para múltiples drones, utilizando programación multithread y mecanismos de sincronización (mutex). La clase Dron representa un dron que interactúa con dos zonas (izquierda y derecha), accediendo a ellas de forma segura mediante exclusión mutua.

Estructura del ejercicio

El ejercicio está dividido en tres carpetas:

- headers: contiene el archivo header dron.hpp con la declaración de la clase Dron junto a sus atributos y sus métodos.
- src: contiene la implementación dron.cpp correspondiente al header, donde se implementa la lógica de comportamiento del dron, incluyendo el lanzamiento del thread y el proceso de despegue.
- main: contiene el archivo main.cpp, donde se prueba el comportamiento multithread de los drones. Allí se crean los mutex que representan las zonas de vuelo, se instancian múltiples drones y se inicia su ejecución mediante los métodos start() y join().

Clases

Dron: Esta clase encapsula un (thread) que ejecuta el método privado takeoff(), responsable de simular el vuelo del dron. Durante su ejecución, el dron intenta ingresar a dos zonas críticas: una zona izquierda y una zona derecha, representadas mediante objetos mutex compartidos entre todos los drones. Para acceder a cada una de estas zonas, el dron debe adquirir el lock correspondiente, lo que garantiza que ningún otro dron pueda ingresar a esa misma zona al mismo tiempo.

Atributos

- int id: identificador único del dron.
- mutex& leftZone: referencia a un mutex que controla el acceso a la zona izquierda.
- mutex& rightZone: referencia a un mutex que controla el acceso a la zona derecha.
- thread thread: thread que ejecuta el comportamiento del dron de forma concurrente.

Métodos

- Dron(int id, mutex& leftZone, mutex& rightZone): constructor que inicializa el dron con su identificador y referencias a los mutex de ambas zonas.
- void start(): lanza el thread de ejecución del dron.
- void join(): espera que el thread del dron finalice.
  ~Dron(): destructor que asegura la limpieza de recursos.
- void takeoff(): método privado ejecutado dentro del thread; contiene la lógica de vuelo del dron y su interacción con las zonas.

Este diseño permite ejecutar múltiples drones simultáneamente, cada uno gestionando su acceso seguro a zonas compartidas del entorno.

Main

El archivo main.cpp crea una simulación en la que múltiples drones (instancias de la clase Dron) son lanzados en paralelo. Cada uno opera su propio thread, accediendo de manera segura a las zonas compartidas:

- Se crean mutex que representan zonas del espacio.
- Se instancian varios objetos Dron, compartiendo los mutex correspondientes.
- Se inicia cada dron con start() y luego se sincronizan con join() para esperar la finalización de todos.

Este enfoque permite observar el comportamiento sincronizado de los threads, previniendo condiciones de carrera gracias al uso correcto de mutex.

Implementación

Durante el ejercicio se buscó simular el comportamiento de varios drones que operan en un entorno compartido. Para lograrlo, se utilizó programación multithreading junto con mecanismos de sincronización como mutex. La intención principal fue modelar cómo

múltiples entidades (drones) pueden interactuar con recursos compartidos (las zonas de vuelo) de forma segura y controlada, evitando conflictos.

Para la implementación se usó mutex que permite un acceso exclusivo a los recursos críticos, evitando errores de concurrencia. Al estructurar esta lógica dentro de una clase, se favorece la reutilización del código, se mejora la organización del programa y este enfoque permite extender el modelo hacia escenarios más complejos, con más drones o más zonas críticas. Además, el diseño respeta el ciclo de vida de los threads, asegurando que todos finalicen de forma correcta y ordenada, sin dejar procesos colgados ni provocar comportamientos inesperados.

En definitiva, este sistema representa una solución clara y funcional al problema de sincronización de múltiples agentes, ofreciendo una base sólida para el desarrollo de sistemas concurrentes, entornos distribuidos o simulaciones de control de tráfico aéreo autónomo.

3. Sistema de Monitoreo y Procesamiento de Robots Autónomos.

Este último ejercicio implementa un sistema de generación y procesamiento de tareas en un entorno industrial simulado. El objetivo principal es modelar la interacción entre múltiples sensores que generan tareas y múltiples robots que procesan dichas tareas.

Estructura del ejercicio

El ejercicio está dividido en un solo archivo:

- main.cpp:

Estructura del código

El programa se compone de tres componentes principales:

- Sensores (productores): generan tareas con descripciones aleatorias y las insertan en una cola compartida.
- Robots (consumidores): retiran tareas de la cola y las procesan.
- Cola compartida: almacena las tareas generadas por los sensores y sirve como punto de sincronización entre sensores y robots.

Las tareas se almacenan en una queue<Tarea> protegida por un mutex (queueMutex) para evitar accesos simultáneos. La condición taskAvailable permite que los robots esperen de forma eficiente a que haya tareas disponibles.

Comportamiento de los sensores

Cada sensor ejecuta una función que simula la creación de tareas a intervalos fijos (175 ms). Cada tarea generada recibe un identificador único (globalTaskId) y una descripción seleccionada aleatoriamente de un vector predefinido. Una vez creada, la tarea se encola y se notifica a los robots mediante taskAvailable.notify_one().

Cuando un sensor termina de generar todas sus tareas, decrementa el contador global activeSensors. Si es el último sensor en terminar, actualiza la bandera allSensorsDone y notifica a todos los robots para que puedan verificar si deben finalizar su ejecución.

Comportamiento de los robots

Los robots ejecutan un ciclo que espera a que haya tareas disponibles. Usan condition_variable::wait() para bloquearse eficientemente hasta que se encole una tarea o todos los sensores hayan terminado y la cola esté vacía. Cuando una tarea es obtenida, el robot simula su procesamiento durante 250 ms, y luego imprime un mensaje indicando que la tarea fue completada. Una vez que no quedan tareas por hacer y todos los sensores finalizaron, el robot sale del ciclo y termina su ejecución.

Consideraciones técnicas

El sistema emplea varios mecanismos de sincronización para evitar errores comunes en entornos multihilo:

- Mutua exclusión: queueMutex protege el acceso concurrente a la cola y a variables compartidas.
- Comunicación eficiente: condition_variable permite que los robots esperen sin consumir CPU hasta que haya trabajo disponible.
- Seguridad de consola: consoleMutex garantiza que los mensajes en pantalla no se mezclen entre hilos.

Además, el diseño evita condiciones de carrera y finalizaciones prematuras gracias a una correcta gestión del ciclo de vida de sensores y robots, respetando la coordinación entre productores y consumidores.

4. CMake y compilación

Para compilar y gestionar nuestros programas de forma robusta y portable, utilizamos CMake como sistema de construcción. A continuación, se explica cada sección del archivo CMakeLists.txt correspondiente a los diferentes ejercicios.

Ejercicio 1

- cmake_minimum_required(VERSION 3.10): Establece que se requiere al menos la versión 3.10 de CMake.
- project(ej1): Define el nombre del proyecto.
- set(CMAKE_CXX_STANDARD 17): Indica que se usará el estándar C++17, que incluye mejoras relevantes como optional, lambdas más potentes, etc.
- include_directories(headers): Añade la carpeta headers/ al path de búsqueda de encabezados (.h). Así, en el código se puede escribir #include "MiClase.h" directamente.
- file(GLOB ...): Busca automáticamente todos los archivos .cpp dentro de las carpetas src/ y main/. Esto facilita el mantenimiento, ya que no es necesario modificar el CMakeLists.txt al agregar nuevos archivos fuente.
- add_executable(ej1 ...): Crea el ejecutable ej1 con todos los archivos fuente encontrados.

Ejercicio 2

Es similar al cmake del ejercicio 1, pero le agregamos lo siguiente:

- find_package(Threads REQUIRED): Busca la biblioteca de hilos del sistema.
- target_link_libraries(ej2 Threads::Threads): Enlaza explícitamente la biblioteca de hilos al ejecutable ej2. Esto es fundamental para que funciones como thread, mutex y condition_variable se resuelvan correctamente en el enlazado.

Ejercicio 3

Este caso es más simple que los anteriores:

- No hay carpetas src/ ni headers/. Solo hay un main.cpp en el mismo directorio que el CMakeLists.txt.
- target_compile_options(...): Añade advertencias del compilador que ayudan a detectar errores y malas prácticas. Esto lo hacemos en el resto de los CMakes.

Compilación y Ejecución

Una vez explicado lo que hace cada parte dentro de cada CMake file, procedemos a explicar cómo se compila y ejecuta los programas.

- cd ej1 # o ej2, ej3
- cd build
- cmake ..
- make
- ./ej1 # o ./ej2, ./ej3 según el proyecto

5. Warnings

Los diferentes programas se ejecutan sin ningún tipo de warning.

6. Conclusión

A lo largo del desarrollo del Trabajo Práctico N.º 2 se logró integrar de manera efectiva conceptos fundamentales de la programación orientada a objetos y la programación concurrente en C++, aplicándolos a problemas concretos y representativos de escenarios del mundo real. Cada uno de los ejercicios abordó distintos enfoques y técnicas, desde la organización y persistencia de datos complejos en el sistema Pokédex, hasta la sincronización de procesos paralelos en entornos simulados como el hangar de drones y el sistema de monitoreo industrial con sensores y robots.

En el caso del sistema Pokédex, se destacó el uso eficiente de estructuras de datos como unordered_map junto con clases bien encapsuladas, validaciones robustas y persistencia binaria, logrando una solución extensible y con buenas prácticas de diseño. El control de drones, por otro lado, sirvió para introducir y consolidar el manejo de thread y mutex, demostrando cómo sincronizar correctamente múltiples agentes que acceden a recursos compartidos. Finalmente, el sistema de sensores y robots llevó estos conceptos al siguiente nivel, con un enfoque productor-consumidor clásico que incluyó coordinación eficiente mediante condition_variable y control detallado del ciclo de vida de los hilos.

El uso de CMake permitió mantener una configuración de compilación organizada, portable y escalable. Cada ejercicio se estructuró con claridad en carpetas separadas y con CMakeLists adaptados a sus necesidades particulares, permitiendo una gestión flexible del código fuente y sus dependencias.

En conjunto, este trabajo representó una experiencia completa en la aplicación de paradigmas clave del lenguaje C++, con foco tanto en la claridad del diseño como en la eficiencia de ejecución. Las soluciones propuestas son funcionales, seguras y fácilmente ampliables, y sientan una base sólida para el abordaje de sistemas más complejos en futuras etapas de formación o proyectos profesionales.