Trabajo Práctico 2

Agustina Barrena, Pilar Otero

abarrena@udesa.edu.ar, potero@udesa.edu.ar

2do Año Ingeniería en Inteligencia Artificial - Grupo 1 - Departamento de Ingeniería,

Universidad de San Andrés

INTRODUCCIÓN

En este informe, nos encargaremos de explicar los diferentes aspectos del trabajo, dejando plasmado su desarrollo y composición. A su vez, se encontrarán con la forma de compilación de los ejercicios y la explicación del contenido de cada una de las carpetas.

DESARROLLO

EJERCICIO 1 – Pokedex

En este ejercicio se desarrolla la implementación de una "enciclopedia" de Pokemones, contando cada uno con sus diversas características. Así, nos encontramos con:

Structs

- `PokemonHash`: implementación de la funcion de hash mediante la sobrecarga del operador () (paréntesis) para la futura implementación del `unordered-map` en la clase `Pokedex`.

Clases

- `Pokemon`: atributos vinculados con el nombre y la experiencia del personaje, metodos de carga y descarga de la información a archivo binario. Cuenta también con la sobrecarga del operador aritmético == necesaria para el desarrollo del `unordered-map` en `Pokedex`.
- `PokemonInfo`: atributos del `Pokemon` (tipo, descripción, ataques disponibles por nivel y la experiencia por nivel), método para la carga de estos a archivo binario, en conjunto con el método de obtención de dicha información para la reconstrucción de estas características del Pokemon.
- `Pokedex`: implementación del `unordered-map` con su clave siendo un objeto `Pokemon` y valor, la información del mismo (objeto `PokemonInfo`). Cuenta también, con funciones de subida y descarga de información a un archivo binario, basándose en las implementadas en las clases que confirman los pares key-value.

Métodos

1. Pokemon:
    - Getters: `getNombre() const, getExperiencia() const`
    - Metodos: `operator==(Pokemon&), cargarInfo(ofstream&), descargarInfo(ifstream&)`
2. PokemonInfo:
    - Getters: `getTipo() const, getDescripcion() const, getAtaquesDisponiblesPorNivel() const, getExperienciaProximoNivel() const`
    - Setters: `setTipo(int)`
    - Metodos: `cargarInfo(ofstream&), descargarInfo(ifstream&)`
3. Pokedex:
    - Metodos: `mostrar(Pokemon&) const, mostrarInfo(Pokemon&) const, mostrarTodos() const, agregarPokemon(Pokemon&, PokemonInfo&), cargarInfo() const, descargarInfo().`

Serialización y Deserialización

Se realiza el guardado de la `Pokedex` en un archivo binario dentro de la carpeta /bin. El proceso incluye guardar cantidad de elementos (`size_t`), serializar cada objeto `Pokemon` y `PokemonInfo` y leer en el mismo orden para reconstruir el `unordered_map`.

Esto permite guardar automáticamente toda la información de los Pokemones en un archivo, sin necesidad de ingresarlos manualmente cada vez que se ejecuta el programa.

Durante la ejecución, al llamar a `descargarInfo()`, el programa abre el archivo binario correspondiente, lee el contenido previamente almacenado en el binario, y reconstruye el mapa `unordered_map` con sus claves (`Pokemon`) y valores

(`PokemonInfo`). Así, se logra recuperar el estado completo de la `Pokedex` tal como fue cargado, permitiendo continuar trabajando con los datos.

Esto representa una ventaja significativa en términos de rendimiento, ya que automatiza la gestión de datos y evita la pérdida de información entre ejecuciones.

EJERCICIO 2 – Control de aeronave en Hangar automatizado

En el caso de este punto del trabajo, nos encargamos de la automatización del despegue sincronizado de cinco drones ubicados en un círculo dentro de un hangar, asegurando que sus zonas adyacentes no constituyan una interferencia. Siguiendo esto, se tiene:

Clases

- Hangar: dentro de sus atributos nos encontramos con la cantidad de drones, un *mutex* con las posiciones (de cada uno de estos), un vector de *threads* (drones) y, por último, un *mutex* que habilita la impresión de los mensajes por consola. A su vez, contiene los métodos para el despegue y la simulación de estos.

Métodos

- `despegar(int), simularDespegues().`

Para la realización de este ejercicio se utilizó `lock_guard` con el objetivo de bloquear el *mutex* que habilita la impresión de los mensajes por consola, evitando así la superposición de los mismos, desbloqueándose de forma automática al salir del scope donde se definieron. A su vez, `defer_lock` ha permitido bloquear las zonas adyacentes (representadas por un *mutex*) al dron que despegará, permitiendo hacer manualmente el bloqueo.

EJERCICIO 3 - Sistema de monitoreo y procesamiento de robots autónomos

El sistema, en el caso de este ejercicio, simula una planta industrial automatizada donde múltiples **sensores** generan tareas de inspección y un conjunto de robots autónomos las procesan. Para lograr esto, hemos desarrollado lo siguiente:

Structs

- Tarea: contiene los atributos requeridos para la identificación de una tarea particular.

Funciones

- `sensor(int), robot(int)`

Para lograr el funcionamiento del código hemos implementado, en este único caso, variables globales (`tareas, mtx, mtxImpresion, cv, terminado, tareasCompletadas`) con el objetivo de utilizarlas en las diversas funciones.

ARCHIVOS

Dentro de la carpeta de los ejercicios 1 y 2 se incluyen 3 nuevas subcarpetas principales:

1. `example`: contiene el archivo main.cpp donde se realizan las pruebas de la simulación correspondiente a la consigna.
2. `scr`: en esta carpeta se encuentran los archivos fuente (`.cpp`) donde se desarrollan las funcionalidades de cada simulación. En particular:
   - Punto 1: implementación de las clases Pokemon, PokemonInfo y Pokedex dentro de los archivos `pokemon.cpp`, `pokemonInfo.cpp` y `pokedex.cpp`.
   - Punto 2: `simulador.cpp`, donde, como en el caso anterior, se desarrollaron los metodos de la clase Hangar.

3. `include`: contiene los archivos de encabezado (`.h`) donde se definen las clases utilizadas, especificando sus atributos y métodos (comportamientos).
4. `build`: archivos necesarios para la compilación de la simulación.

Por su parte, en el punto 3 solamente se encuentran la carpeta `build`, que contiene al binario para la compilación del proyecto y la carpeta `example`, donde se encuentra el archivo `main` en el que se implementan todas las funciones necesarias.

En adición, todos los ejercicios también presentan el archivo CMakeLists.txt, con las instrucciones para la configuración y compilación del proyecto.

COMPILACIÓN

Para la compilación del proyecto hemos optado por utilizar la herramienta *CMake*, facilitando la generación de los archivos de compilación. De esta forma, cada ejercicio, tal como mencionamos previamente, cuenta con su propio archivo CMakeLists.txt.

A continuación, se detallan los pasos a seguir:

1. En caso de no contar con la herramienta *CMake* descargarla desde https://cmake.org/download/ o escribiendo en la terminal el código correspondiente a su sistema operativo:
   MacOs: `brew install cmake`
   Linux (Ubuntu): `sudo apt update`
   `sudo apt install cmake`
2. Crear la carpeta de compilación ejecutando:
   ```
   cmake -S . -B build
   ```
3. Compilar el proyecto:
   ```
   cmake --build build
   ```

4. Ejecutar el programa:

   ```
   ./build/bin/nombreDelEjecutable
   ```

   Siendo el nombre del ejecutable `projectX`, reemplazando X por el número de ejercicio (1,2 o 3).

Observaciones

1. En caso de copiar y pegar el coigo de compilación, hacerlo sin agregar espacios ya que de otra manera no funcionará la compilación del archivo. Recomendamos escribirlo manualmente.
2. NO abrir este PDF desde el IDE utilizado ya que al copiar el código de compilación se agregarán espacios que harán que esto no funcione. Abrir desde un navegador o, igualmente adjunto el Word a la carpeta.

CONCLUSIÓN

En conclusión, este trabajo nos permitió consolidar en profundidad los conceptos clave de la materia. El desarrollo de las clases, en el caso del ejercicio uno, no solo facilito los procesos de carga y descarga de la información a un archivo binario de las diferentes clases polimórficas, sino que favoreció también una comprensión más sólida de la implementación de estos métodos. En adición, hemos ahondado en el uso de *threads*, lo cual nos permitió apreciar las grandes ventajas que ofrece la concurrencia al abordar problemas de mayor complejidad, destacando su utilidad en la construcción de programas aún más complejos.