Trabajo Práctico 2

Paradigmas de Programación

Pereyra Agustín[1], **Pizarro Federico**[2]

Universidad de San Andrés, Provincia de Buenos Aires, Argentina

Ingeniería en Inteligencia Artificial

[1] *apereyra@udesa.edu.ar*

[2] *fpizarrodalmaso@udesa.edu.ar*

19 de junio de 2025

Este trabajo tiene el objetivo de afianzar los conceptos de *Multithreading* o programación multihilo, con sincronización y uso de datos compartidos como también serialización, uso de clases y structs, containers y streams en C++ a partir de diferentes ejercicios prácticos.

Aclaración: *Para cada ejercicio se diseñó un archivo Makefile para automatizar la compilación y ejecución con g++ de cada uno de los ejercicios, utilizando las flags −Wall −Wextra, −Wpedantic. En cada sección se detallan los comandos correspondientes para cada uno*

1. Pokedex

Esta ejercitación pide desarrollar un pokedex digital funcional en C++. Donde se implementaron tres clases Pokemon, PokemonInfo y Pokedex.

En la clase PokemonInfo, se debía decidir que contenedores utilizar para attacksByLevel y nextLevelExperience, en ambos decidimos usar un array de 3 elementos, ya que siempre contamos con 3 niveles. Esto nos permite guardar menos información en la serialización, ya que son de largo fijo, esto se refleja en un menor peso en el archivo donde guardamos los datos, en conclusión esto permite que el proyecto sea aún más escalable.

Luego, para resolver el ejercicio b se implementó un itemB.cpp donde se generán los pokemones pedidos y luego se utiliza una función Pokedex::showAll(), la cual usa únicamente las instancias de Pokemon para buscar y mostrar la información de los tres pokemones.

En el ejercicio c se pide probar el funcionamiento de una función show, la cual permita mostrar la información de un pokemon en particular guardado en el pokedex, solamente con la instancia de Pokemon. Para esto se creó un itemC.cpp donde se muestra un caso en el que si existe el pokemon y otro donde no existe.

Por último para la consigna opcional del ejercicio d, en itemD.cpp se implemtó un menú para poder controlar un pokedex, donde se puede guardar pokemones, mostrar uno o varios pokemones, guardar el pokedex, y cargar un pokedex desde un archivo. La idea principal es poder serializar y deserializar todos los elementos del pokedex, es por esto que las clases Pokemon y PokemonInfo, cuentan con estos métodos.

1.1. Pokemon

La clase Pokemon cuenta con los atributos privados nombre, y nivel de experiencia, el cual es un entero positivo.

Implementa métodos constructores, por defecto, con el nombre y también otro con la experiencia, además un destructor por defecto. Tiene los getters y setters básicos, funciones de serialización y deserialización. Además, se sobrecarga el operador <<, para imprimir en consola.

Se sobrecarga el operador == para poder utilizar la función . find () de los unordered_map para comparar las claves, ya que se guardan como la key del mapa. Por último, especializamos la plantilla std :: hash<Pokemon>, lo cual permite al mapa hashear el objeto basándose en el nombre del pokemon.

1.2. PokemonInfo

Se implementaron los atributos privados type y description , los cuáles son string que representan que tipo de pokemon es, y una breve descripción. Además, PokemonInfo cuenta con un unordered_map<string, damage_t>, donde damage_t es un unsigned int , el cuál representa cuánto daño causa cada ataque. Por último, también cuenta con un vector<experience_t>, donde experience_t es también un unsigned int , en él se guarda la experiencia necesaria para alcanzar el próximo nivel.

Esta clase cuenta con un constructor por defecto, y otro con todos los atributos pasados como argumentos, y también un destructor por defecto. Implementa getters básicos de todos los atributos, y funciones de serialización y deserialización. Por último cuenta con una sobrecarga en el operador << para imprimir la información por consola.

1.3. Pokedex

En esta clase implementamos dos atributos privados, un unordered_map<Pokemon, PokemonInfo> donde guardamos cada pokemon como clave, con su respectiva información como valor. Además, un string donde se almacena el nombre del archivo donde se guarda la información del pokedex, esto siempre se guarda en la carpeta data .

Tiene sus constructores básicos, uno de ellos permite definir en que archivo se guardará la información, y un destructor por defecto. Luego cuenta con un método para corroborar si está vacío y otro para agregar un pokemon al pokedex. Por otro lado se implementan los métodos para mostrar uno o varios pokemones, y los métodos para guardar la información del pokedex en un archivo y también cargar al pokedex la información desde uno.

1.4. Compilación y Ejecución

Posicionandose dentro de la carpeta exercise1 :

I. make b: compilar ejercicio b
II. make run−b: compilar y ejecutar ejercicio b
III. make c: compilar ejercicio c
IV. make run−c: compilar y ejecutar ejercicio c
V. make d: compilar ejercicio d
VI. make run−d: compilar y ejecutar ejercicio d
VII. make d−valgrind: ídem d con Valgrind
VIII. make run−d−valgrind: ídem run-d con Valgrind
IX. make all : compila los tres ejercicios
X. make run−all: compila y ejecuta los tres ejercicios
XI. make clean: eliminar ejecutables y guardados

2. Control de Aeronave en Hangar Automatizado

Para que cada Dron espere a que sus zonas laterales estén libres para poder despegar, se utilizaron punteros de std :: mutex como atributos de la clase Dron, de manera tal que cada dron tiene los mutex de sus 2 zonas laterales, e intenta bloquearlos para que le permita despegar. Si algún otro dron tiene lockeado al menos 1 de sus zonas este tendrá imposibilitado el despegue y quedará a la espera de que se liberen.

2.1. Dron

Como ya fue mencionado contiene como atributo 2 punteros 2 mutex, representando sus 2 zonas laterales. Además tiene un ID para identificar al mismo.

Los métodos implementados son su constructor, un logger que se usa de forma privada dentro del de despegue, y takeOff () . Esta última simula la situación de despegue de un Dron, esperando la liberación de sus zonas laterales, el cual se logra con un multiple−lock de los mutex respectivos a cada zona lateral así evitando un posible *deadlock*. Además, una vez ya habiendo despegado, se aparenta el tiempo de demora al alcance de los 10m necesarios para habilitar a los drones cercanos poder despegar.

2.2. Hangar

La situación del Hangar se representa en el código principal, en donde se inicializan tanto las zonas como los drones y se inicia un std :: thread por cada uno de los drones. Luego se asegura que el programa espere a que terminen de ejecutarse por completo los threads antes de que se liberen los punteros a las zonas (mutex).

2.3. Compilación y Ejecución

Posicionandose dentro de la carpeta exercise2 :

I. make main: compilar
II. make run: compilar y ejecutar
III. make run−valgrind: compilar y ejecutar con Valgrind
IV. make clean: eliminar ejecutables

3. Sistema de Monitoreo y Procesamiento de Robots Autónomos

Para la resolución de esta parte, se hizo uso de una sola estructura llamada Task, la cuál tiene las propiedades de las tareas emitidas por los Sensores y procesadas por los Robots, ambos implementados como hilos (threads). Además está casi todo el código dentro del mismo archivo source, main.cpp, y lo único fuera de él es tanto la implementación como la definición de la estructura.

El struct Task solo contiene el ID del mismo, el ID del sensor que lo generó y una descripción del mismo, la cual se elige aleatoriamente de un array de descripciones genéricas. Además tiene solo un constructor que tiene como función adicional, simular el delay de 175ms de la creación de una tarea.

Cuando el programa comienza, se definen tanto las constantes como las variables globales, entre ellas podemos destacar a los 2 objetos std :: mutex, encargados de sincronizar tanto el uso de la consola como de la std :: queue de tasks, al objeto std :: atomic que lleva la cuenta de los sensores activos y a la variable std :: condition_variable usada para sincronizar a los sensores con los robots.

Luego se inician los threads definidos como std :: jthreads para que todos sean ejecutados por completo y no puedan ser interrumpidos. Luego el funcionamiento se rige por la siguiente estructura para cada hilo:

3.1. Sensores

I. Por cada tarea a generar
    I. Se bloquea el std :: mutex de la queue para crear la tarea, asignando los atributos y agregándola a la cola.
    II. Se bloque la consola y se hace el log respectivo.
    III. Se notifica a los robots (que están en espera) que hay una tarea disponible mediante la variable condicional.
II. Lockeando el mutex destinado a los sensores, se actualiza el contador atómico y se verifica si es el último sensor. Si es el caso se notifica a los robots.

3.2. Robots

I. Mientras que hayan sensores activos o tareas disponibles
    I. Espera hasta ser notificado por los sensores que hay una tarea nueva disponible, o que terminó el último sensor
    II. Se quita la tarea de la cola y se la procesa (esperando el respectivo acceso mediante el mutex)
    III. Se simula el delay del procesamiento de la tarea de 250ms
    IV. Se hace el log del procesamiento bloqueando primero la consola.
II. Se loggea la finalización del trabajo del Robot

3.3. Compilación y Ejecución

Posicionandose dentro de la carpeta exercise3 :

I. make main: compilar
II. make run: compilar y ejecutar
III. run−valgrind: compilar y ejecutar con Valgrind
IV. make clean: elimina ejectuables

Warnings: *Para cada uno de los apartados fueron solucionados tanto los errores, warnings como recomendaciones indicadas por el compilador g++ dadas las flags mencionadas anteriormente.*