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*

12 de junio de 2025

Este trabajo tiene el objetivo de afianzar los conceptos de *Multithreading* o programación multi-hilos, 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

1.1. Pokemon

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

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. Además, especializamos la plantilla `std::hash<Pokemon>`, lo cual permite al mapa hashear el objeto basándose en el nombre del Pokemon.

Implementa métodos constructores, por defecto, con el nombre y también otro con la experiencia. Tiene los getters y setters básicos, funciones de serialización y deserialización. Por último, se sobrecarga el operador , de forma que se pueda imprimir en consola.

1.2. PokemonInfo

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 lockearlos 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 y a la variable condicional `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 lockeando 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

Aprendizaje