Universidad de San Andrés

Paradigmas de Programación (I102)

Trabajo Práctico 2

Garcia Giacchetta, Alejandro
Puentes, Santiago Martin

Cuatrimestre 1 – Año 2025
19 de junio

Resumen

Este informe documenta las soluciones propuestas para los tres ejercicios exigidos en el Trabajo Práctico 2 de la cátedra I102 (Paradigmas de Programación) de la Universidad de San Andrés. Se incluyen detalles de la compilación de cada programa, los comandos utilizados, y una sección de desarrollo de cada ejercicio para las explicaciones de diseño, implementación y resultados.

Índice

1. Introducción General

Este Trabajo Práctico se compone de tres ejercicios independientes, cuya resolución completa el aprendizaje sobre contenedores STL, hashing, programación multihilo y concurrencia.

Para introducir los ejercicios en cuestión:

1. **Pokedex:** implementar un contenedor `Pokedex` que asocie objetos `Pokemon` a `PokemonInfo` usando `std::unordered_map` con hash propio y operadores necesarios para usar `Pokemon` como clave.

2. **Control de Aeronave en Hangar Automatizado:** simulación multihilo de cinco drones que deben adquirir *exclusivamente* las dos zonas adyacentes (izquierda y derecha) antes de despegar, para evitar interferencias.

3. **Sistema de Monitoreo y Procesamiento de Robots Autónomos:** sensores que generan tareas periódicas y robots que consumen estas tareas desde una cola compartida, usando `std::thread`, `std::mutex`, `std::condition_variable`, y `std::queue`.

Cada ejercicio en su propia carpeta mantiene una estructura general de source file que incluye las implementaciones específicas de cada ejercicio facilitadas en un único `.cpp` y las declaraciones generales en un único header de la carpeta include.

```
TP2_Garcia_Giaccheta_&_Puentes/
├─ EJ_1/
│   ├─ include/
│   │   └─ poke.hpp
│   ├─ src/
│   │   └─ poke.cpp
│   ├─ main.cpp
│   └─ Makefile
├─ EJ_2/
│   ├─ include/
│   │   └─ drone.hpp
│   ├─ src/
│   │   └─ drone.cpp
│   ├─ main.cpp
│   └─ Makefile
├─ EJ_3/
│   ├─ include/
│   │   └─ events.hpp
│   ├─ src/
│   │   └─ events.cpp
│   ├─ main.cpp
│   └─ Makefile
└─DOCUMENTATION.pdf
```

2. Requisitos Compilación

Para todos los ejercicios se utilzaron instrucciones de compilación y ejecución comunes gracias a Makefile.

- Compilador: `g++` (o `clang++`) con soporte de C++23. Los ejercicios se compilan usando `Makefile` clásicos compatibles con `GNU Make 3.81` o superior.

- Comando de compilación dentro de cada Makefile.

```
-std=c++23 -Iinclude/.
           -Wall -Wextra -Wpedantic -Werror \
           -Wconversion -Wsign-compare -Wshadow -Wpointer-arith \
           -Wfloat-equal -Wcast-align -Wformat -Wmissing-declarations \
           -Wunreachable-code -Wundef -Winline -Wunused \
           -Wnon-virtual-dtor -Wduplicated-cond -Wduplicated-branches \
           -Wlogical-op -Wnull-dereference -Wdouble-promotion -Wuseless-cast \
           -Wzero-as-null-pointer-constant
```

- Para compilar y ejecutar cada ejercicio basta ejecutar el comando `make -C EJ_{Número de ejercicio sin {}}` estando parado en el directorio del TP.

Advertencias de compilación

El proyecto se compila con una batería extensa de banderas de advertencia que permiten detectar errores sutiles en tiempo de compilación, algunos de los cuales detallamos a continuación.

- `-Wall`: Habilita las advertencias básicas del compilador (uso de variables no inicializadas, expresiones sospechosas, etc.).
- `-Wextra`: Añade advertencias adicionales, como parámetros sin usar o conversiones implícitas potencialmente peligrosas.
- `-Wpedantic`: Verifica conformidad estricta con el estándar del lenguaje.
- `-Wconversion`: Avisa cuando hay conversiones entre tipos que podrían provocar pérdida de datos (por ejemplo, de `int` a `char`).
- `-Wshadow`: Detecta cuando una variable local oculta otra del mismo nombre en un ámbito externo.
- `-Wsign-compare`: Avisa al comparar `signed` con `unsigned`, posible fuente de errores lógicos.
- `-Wunreachable-code`: Detecta bloques de código que no se pueden ejecutar.

Durante el desarrollo se resolvieron los conflictos de tipo y ambigüedad de nombres sugeridos por los warnings. El código final compila sin advertencias, lo que refuerza su robustez.

3. Ejercicio 1. Pokedex

3.1. Objetivo y Descripción

Este ejercicio propone la implementación de un sistema tipo Pokedex digital utilizando objetos y contenedores STL. El sistema debe representar información estructurada sobre distintos Pokemon, clasificarlos por tipo y permitir su consulta mediante operaciones de búsqueda. Para ello, se diseñan tres clases principales: `Pokemon`, `PokemonInfo` y `Pokedex`.

- **Pokemon**: representa al Pokemon individual e incluye los atributos `nombre` (una cadena de texto) y `experiencia` (un entero positivo). Estos datos deben ser accesibles para su visualización.
- **PokemonInfo**: contiene información específica sobre el Pokemon, como su `tipo` (agua, fuego, planta, etc.), una `descripcion` textual, un conjunto de `ataquesDisponiblesPorNivel` representado como un contenedor asociativo (nombre del ataque y daño), y una lista de `experienciaProximoNivel` para los tres niveles posibles. Se debe justificar la elección de contenedores en el informe.
- **Pokedex**: almacena la información de los Pokemon utilizando un `std::unordered_map`, donde la clave es un objeto `Pokemon` y el valor correspondiente es un objeto `PokemonInfo`. Se requiere que `Pokemon` implemente los operadores necesarios para ser utilizado como clave, y que se defina un functor de `hash` utilizando exclusivamente el atributo `nombre`.

La clase `Pokedex` debe incluir un método `mostrar` que reciba un `Pokemon` y muestre su información si está registrado, o indique `"¡Pokemon desconocido!"` en caso contrario. Además, se debe implementar un método `mostrarTodos` para imprimir la información de todos los Pokemon almacenados.

El ejercicio requiere crear tres Pokemon predefinidos con sus respectivas instancias de `PokemonInfo`, agregarlos al `Pokedex`, e invocar `mostrarTodos`. También se debe probar el método `mostrar` con un Pokemon existente y uno inexistente para verificar su funcionamiento.

Opcionalmente, se propone extender el sistema con soporte de persistencia en archivos. El nombre del archivo debe pasarse como argumento al constructor del `Pokedex`, el cual cargará automáticamente los datos desde el archivo si está disponible, y actualizará el mismo ante cada modificación posterior.

3.2. Desarrollo

De cierta forma el enfoque del desarrollo de la solución viene establecida de forma forma predeterminada por las consignas en cuanto a funcionalidad y clases o estructuras de datos. Sin embargo, algunos detalles quedan por explicar en cuanto a las decisiones que tomamos sobre la implementación elegida. En las siguientes subsecciones se explica cada apartado al detalle. Una aclaración general importante es que se hace uso del artilugio `uint` el cual no es más que un `unsigned int` que facilita los usos de enteros positivos (dado que todos los enteros que se manejan son possitivos).

Clase `Pokemon`

No tiene mucha complejidad, se definen constructor default (como en todas las clases implementadas) para poder instanciar la clase desde contenedores o formas genéricas sin especificar los atributos. Lo interesante es la sobrecarga del `==` que permite la comparación entre pokemones. Esta verifica que los nombres sean iguales. Fue pensado en función de que el hash de cada Pokemon se hace en base a su nombre, por lo que la comparación debería ser de la misma forma.

```
1 class Pokemon {
2     string nombre;
3     uint experiencia;
4     public:
5         Pokemon() = default;     // Para estructuras que inicialicen el
              objeto default
6         // Constructor general
```

```
        Pokemon(const string& name, uint xp) : nombre(name),
            experiencia(xp) {};
        // Devuelve el nombre del Pokemon
        const string& getNombre() const {return nombre;}
        // Devuelve el xp del Pokemon
        uint getXP() const {return experiencia;}
        // Permite la comparación entre dos objetos Pokemon mediante el
            nombre
        bool operator==(const Pokemon& other) const {return (nombre ==
            other.nombre);}
        ~Pokemon() = default;    // Destrucción default
};
```

Algo importante por mencionar es la sobrecarga del functor para la utilización de Pokemon como clave del `unordered_map` que se usa en la pokedex. La especialización está en el archivo de encabezado porque el compilador necesita verla en cualquier lugar donde se utilice un unordered_map con Pokemon como clave. Las especializaciones de plantillas no se pueden separar entre declaración e implementación. Si se define solo en un archivo fuente, otros archivos no sabrán cómo generar el hash y la compilación fallará.

```
namespace std {
    template<>
    struct hash<Pokemon>{
        size_t operator()(const Pokemon& p) const {
            return hash<string>()(p.getNombre());   // Se define el functor
                como el retorno de un hash del nombre del pokemon
        }
    };
}
```

Clase PokemonInfo

Sobre esta clase si es importante mencionar los contenedores utilizados. Primero, se especifica que cada Pokemon cuenta con hasta 3 posibles niveles, lo cual posibilita el uso de array. Esto fue motivada por la simpleza de la estructura y fácil mantenimiento siempre y cuando el tamaño sea fijo como en este caso. Por otro lado, la consigna no habla de la cantidad de ataques que cada Pokemon tiene disponibles, por ende asumimos un tamaño de contenedor indeterminado y variable (si es que se quisieran insertar más ataques una vez desarrollado, habría que implementar el método que lo permita). Entonces, la decisión fue usar `vector`. Además de la simpleza del uso en vector, es posible recorrerlo de forma iterada y dado que lo único importante es imprimir todos los ataques no implica un costo significativo (con cualquier estructura habrá O(n)). Se podría implementar una utilidad de ordenamiento por efectividad con facilidad si es que se desea a futuro. Los miembros de la clase son bastante claros (getters comunes que devuelven referencias constantes para evitar modificaciones), como puede verse en la implementación del código.

```
class PokemonInfo {
    string tipo;                        // Tipo del pokemon
    string descripcion;                 // Descripción del pokemon
    set_ataques ataquesPNivel;          // Ataques del pokemon
    array<uint, 3> experienciaPNivel;   // Experiencia para llegar a cada
        nivel
    public:
        PokemonInfo() = default;      // Para estructuras que inicialicen el
            objeto default
        // Constructor general de la clase
        PokemonInfo(const string& t, const string& d, const set_ataques& a,
            const array<uint, 3>& xp) :
                tipo(t), descripcion(d), ataquesPNivel(a),
                    experienciaPNivel(xp) {}
```

```
        // Devuelve el tipo del Pokemon
        const string& getTipo() const {return tipo;}
        // Devuelve la descripcion del Pokemon
        const string& getDescripcion() const {return descripcion;}
        // Devuelve los ataques del Pokemon
        const set_ataques& getAtaques_lvl() const {return ataquesPNivel;}
        // Devuelve la experiencia de cada nivel del Pokemon
        const array<uint, 3>& getXP_lvl() const {return experienciaPNivel;}
        ~PokemonInfo() = default;   // Destrucción default
};
```

Clase Pokedex

Esta clase si que trae cierta complejidad. El contenedor fundamental es el `unordered_map`. Algunas cuestiones importantes son la implementación de un string para el nombre del archivo de guardado (adicional de serialización). Entonces, además del constructor default tenemos la sobrecarga que recibe como párametro el nombre del archivo y lo guarda en el miembtro `saveFile` para luego, en su cuerpo, cargar la pokedex guardada mediante el método `deserializar()`. Se implementaron los métodos de muestra de pokemones y los extras de serialización que explicaremos a continuación.

```
class Pokedex {
    unordered_map<Pokemon, PokemonInfo> Pokedata;   // Dex con datos de
       pokemones descubiertos
    string saveFile;        // Nombre del archivo para serialización

    public:
        Pokedex() = default;        // Para inicialización default

        /*
        * Constructor específico para iniciar la pokedex en base a un
           archivo binario.
        * Se deserializa una vez inicializada la estructura
        */
        Pokedex(string save_file) : saveFile(save_file) {deserializar();}
        // Muestra por pantalla un objeto Pokemon si se encuentra en la
           pokedata
        void mostrar(const Pokemon& poke) const;
        // Muestra todos los pokemones contenidos por pokedata
        void mostrarTodos() const;
        // Permite guardar toda la informació de la pokedex en un archivo
           de salida binario
        void serializar() const;
        // Permite cargar toda la información de una versión de una pokedex
           almacenada en un archivo de entrada binario
        void deserializar();
        // Destructor específico para serializar si hace falta
        ~Pokedex() {
            if(!saveFile.empty()) serializar(); // Si tiene un file para
               serializar entonces se serializa antes de destruir la
               instancia
        }
        // Operador para facilitar la búsqueda e incersión en el mapa
        PokemonInfo& operator[](const Pokemon& poke) {return
           Pokedata[poke];}
};
```

Un detalle en el destructor de la clase, se le agrega la verificación para ver si fue inicializado con un `saveFile`, entonces si el string no está vacío se procede con la serialización (guardado de la pokedex). Esta decisión de implementación nos garantiza que mientras estemos trabajando en el programa el

archivo se mantiene cerrado (solo serializar y deserializar lo abren) y que se hace el backup al final sobre el estado final de la pokedex. No habrá problemas de re escritura de información en el binario ya que se sobrescribe su información. Tampoco es posible que se pierdan datos ya que antes de destruirse la Pokedex o salir de scope el destructor se encargará de la serialización correspondiente. Se podría (de ser necesario) implementar un constructor o método de merge para agregar datos a una pokedex desde otra y tampoco habría problemas de serialización gracias a la implementación base.

Por otro lado tenemos la sobrecarga de los corchetes, lo cual permite hacer uso del mapa por fuera de la clase. La implementación trae versatilidad en el manejo del contenido `Pokedex[Pokemon] = PokemonInfo` permite hacer o insert de forma elegante y legible, exactamente como se puede hacer al mapa por si solo. Sin embargo, es importante tener ciertas precauciones ya que `Pokedex[Pokemon]` de por sí devuelve una referencia del mapa, si Pokemon no se encontraba en el mapa lo agrega bindeado a una instancia default de PokemonInfo (sin atributos inicializados) esto quiere decir que si se quiere ver la PokemonInfo de un default imprimiéndolo por pantalla, no tendremos ningún valor relacionado a un Pokemon. Por esa razón existe el método `Pokedex::mostrar()`. Ahora lo que si facilita es la resignación de datos del Pokemon aunque posiblemente sea inútil dado el contexto de la Pokedex, pero en fin, gracias a esta sobrecarga se puede trabajar de esa forma.

Impresión por pantalla

Los métodos de la pokedex son simples, utilizan el cout para imprimir como puede verse a continuación

```
void Pokedex::mostrar(const Pokemon& poke) const {
    try {
        const auto& search = Pokedata.at(poke); // at() falla si no lo
            encuentra
        cout << "Pokemon encontrado...\n" << poke << "\n" << search <<
            endl;    // Si tiene exito se manda al cout
    }
    catch (const out_of_range&) {cout << " Pokemon  desconocido!" << endl;}
        // Si no lo encuentra se imprime que se desconoce
}

void Pokedex::mostrarTodos() const {
    cout << "Mostrando toda la Pokedex\n";
    for (const auto& [poke, info]:Pokedata) {cout << poke << info << "\n";}
    // Iterando sobre cada pair del mapa se manda al cout por
        desempaquetado de la estructura del pair
}
```

El artilugio de utilizar los `<<` para la impresión directa viene gracias a la sobrecarga de los mismos. En esta implementación se puede apreciar el uso de la técnica de desempaquetado para `pair`.

```
ostream& operator<<(ostream& os, const Pokemon& poke) {
    os << "Nombre: " << poke.getNombre()         // Manda al stream el nombre
        << "\nXP: " << poke.getXP() << endl;     // Manda al stream el xp
    return os;  // Devuelve el stream
}

ostream& operator<<(ostream& os, const PokemonInfo& info) {
    os << "Tipo: " << info.getTipo()     // Manda al stream el tipo
        << "\nDescripción: " << info.getDescripcion() // Manda al stream la
            descripcion
        << "\nAtaques disponibles: [";
    for (const auto& [ataque, daño] : info.getAtaques_lvl()) {
        os << "(" << ataque << ": " << daño << ")"; // Manda al stream los
            ataques pareados con cada nivel de daño
    }
    os << "]\nExperiencia por nivel: ";
```

```
    os << "[";
    for (const auto& xp : info.getXP_lvl()) {
        os << "(" << xp << ")"; // Manda al stream el xp requerida para
            cada nivel
    }
    os << "]" << endl;
    return os;  // Devuelve el stream
}
```

Serialización

Como adicional se agregan las funcionalidades de serialización y deserialización. Ya se explicó lo fundamental de la idea de implementación y la forma en que la Pokedex puede serializar y deserializar en binario de guardado.

Si es importante destacar la implementación de las sigueintes líneas en la serialización.

```
    ifstream in(saveFile, ios::binary);
    if (!in || in.peek() == ifstream::traits_type::eof()) {
        // Si no se pudo abrir manda al stream de errores el aviso de error
            y no continua
        cerr << "Error al abrir el archivo " << "\n";
        return;
    }
```

El extracto es el inicio de la función y lo que le garantiza al programa es no terminar por error o colgarse si es que el archivo no puede abrirse. Lo mismo fue usado en deserializar para verificar si no se puede abrir. Pero esté método tiene una particularidad y es que si el binario de apertura se encuentra vacío hacer el peek y verificar si el primer carácter es el final del archivo, permite que no se cuelgue en la lectura y pueda avanzar aunque será imposible que cargue nada (la pokedex guardada no existe).

Implementación de `main.cpp`

Lo que se agregó al main es sencillamente un ejemplo de uso de las estructuras de la Pokedex y los Pokemon y PokemonInfo tal como se pide en la consigna. En el código se puede ver el uso del artilugio de la sobrecarga de corchetes y los detalles de serialziación y guardado al salir del scope. En la salida por consola se podrá ver como se muestran todos lo pokemones agregados en main, como se guardan y cuando uno desconocido se pide mostrar. Luego se muestra la recuperación desde el binario

4. Ejercicio 2. Control de Aeronave en Hangar Automatizado

4.1. Objetivo y Descripción

Este ejercicio consiste en simular un sistema de despegue coordinado para cinco drones cuadricópteros ubicados en círculo dentro de un hangar automatizado. Cada dron necesita adquirir el control exclusivo de las dos zonas adyacentes (izquierda y derecha) antes de iniciar el despegue, ya que la turbulencia de drones cercanos puede afectar su estabilidad. En total, hay cinco zonas de interferencia, una entre cada par de drones consecutivos.

El comportamiento de cada dron sigue la siguiente lógica: al iniciar, verifica si puede tomar el control de sus dos zonas laterales. Solo podrá hacerlo si ambas están libres o si los drones vecinos ya alcanzaron una altura segura de 10 metros. Si no es posible, el dron queda a la espera hasta que ambas zonas queden disponibles. Una vez que adquiere las zonas, inicia el despegue. Transcurridos cinco segundos, alcanza la altura segura y libera las zonas.

Para modelar este comportamiento se debe implementar una solución en utilizando programación multihilo, empleando exclusivamente `std::mutex` y `std::lock` (incluyendo multilock). Se busca minimizar el tiempo total necesario para que todos los drones completen su despegue, sin utilizar esperas artificiales salvo los cinco segundos reales de despegue. Además, se debe garantizar que no haya superposición de mensajes por consola.

El programa debe incluir una salida controlada que permita observar el estado de cada dron en tiempo real. Por ejemplo:

```
Dron 0 esperando para despegar...
Dron 0 despegando...
Dron 1 esperando para despegar...
Dron 2 esperando para despegar...
Dron 2 despegando...
Dron 3 esperando para despegar...
Dron 4 esperando para despegar...
Dron 0 alcanzó altura de 10m
Dron 4 despegando...
Dron 2 alcanzó altura de 10m
Dron 1 despegando...
Dron 4 alcanzó altura de 10m
Dron 3 despegando...
Dron 1 alcanzó altura de 10m
Dron 3 alcanzó altura de 10m
```

Se deberá validar que los mensajes se imprimen de forma ordenada y sin entrelazamiento, cumpliendo con los requisitos de sincronización y exclusión mutua.

4.2. Desarrollo

Para la resolución de la problemática de este ejercicio se implementó la clase *Drone* que contiene de forma independiente los datos de cada instancia (cada dron) y los métodos de despegue y logeo por consola. Luego mediante threads en *main.cpp* se bindean los objetos dentro del contenedor, que cuando es creado asigna a cada thread un objeto y su método de despegue con los respectivos parámetros para que arranque su trabajo (despegar).

Diseño de la clase Drone

- Cada objeto dron será quien contenga su propia información (`private`) sobre su id (`int id`) y las zonas (`int z1, z2`) que le corresponde usar. Las zonas que debe utilizar se pasan al constructor por parámetros al igual que el id. Estos datos son pasados a la clase por el contructor cuando es instanciada. Además cuenta con un constructor default para ser instanciada por contenedor inicializando y un destructor default.

```
class Drone{
    int id;       //numero de dron
    int z1, z2;  //numeros de zonas que usa
    public:
        Drone() = default;       //constructor para inicializar dentro
            de containers default

        // Constructor de drone, toma las zonas asginadas y el id del
            dron
        Drone(int drone_id, int zona1, int zona2) : id(drone_id),
            z1(zona1), z2(zona2) {}

        // Emula la acción de despegue del dron
        void despegar(mutex& print, array<mutex, 5>& zonas,
            chrono::steady_clock::time_point start);

        // Permite hacer uso de cout como recurso compartido evitando
            race conditions
        void log(string msg, chrono::steady_clock::time_point start,
            mutex& print) const;
        ~Drone() = default;
};
```

Listing 1: Definicion de Drone

- Miembro público + `log`: Cada vez que necesitemos hacer un log de estado (imprimir mensaje de espera, despegue o finalización) se llama la función para dejar un mensaje al cout. De esta forma se puede también, loggear en el tiempo en el que sucede cada evento y tener una noción de la dinámica del sistema.

```
void Drone::log(string msg, chrono::steady_clock::time_point start,
    mutex& print) const {
    auto now = chrono::steady_clock::now();
    auto elapsed = chrono::duration_cast<chrono::milliseconds>(now -
        start).count();

    print.lock();
    cout << "[T: " << elapsed << "ms] " << "Drone " << id << msg <<
        endl;
    print.unlock();
}
```

- Miembro público + `despegar`: Cada vez que se quiera hacer un print por `cout`, se llama al método log. Por otro lado la forma de evitar un *deadlock* por uso de zonas compartidas es mediante `lock(zonas[usa[0]], zonas[usa[1]])` que no es mas que lockear simultáneamente los mutex de las 2 zonas subyacentes al dron (las que usa). Se pausa el hilo por 5000 ms, justo cuando alcanza la altura para liberar las zonas, entonces se avisa que alcanzó altura de 10m y se liberan los mutex.

```
void Drone::despegar(mutex& print, array<mutex, 5>& zonas,
    chrono::steady_clock::time_point start) {
    log(" esperando para despegar...", start, print);
    lock(zonas[z1], zonas[z2]);
    log(" despegando...", start, print);
    this_thread::sleep_for(chrono::milliseconds(5000));
    log(" alcanzó altura de 10m", start, print);
    zonas[z1].unlock(); zonas[z2].unlock();
}
```

- Los objetos `Drone` están contenidos en el main por `array<Drone, 5> drone_o` y se inicializan mediante un for statement. De esta forma se le pasan las zonas correspondientes y su id para cada iteración que representa la creación de cada dron.

Diseño de la lógica multihilo en `main.cpp`

- Se utiliza un mutex individual para cada zona. Todas están contenidas en la estructura `array<mutex, 5>zonas` del main la cual es pasada por referencia a cada instancia de Drone cuando se llame a su método `despegar()`.
- Cada drone es asignado por 2 zonas y se maneja el uso de ellas mediante un `thread` por dron. Los threads son contenidos por `array<thread, 5>drones` e inicializados en el bucle for.
- Finalmente se espera a que cada hilo termine aplicando antes de salir del scope de main.

```
int main() {
    auto start_time = chrono::steady_clock::now(); //Inicializamos el
        cronometro
    mutex print;     //mutex para cout
    array<mutex, 5> zonas;  //arreglo de zonas
    array<Drone, 5> drone_o;     //arreglo de objetos Drone
    for (int i = 0; i < 5; ++i) {
        drone_o[i] = Drone(i, i, (i+1)%5);     //inicializamos cada drone
    }
    array<thread, 5> drones;     //arreglo de threads

    for (int i = 0; i < 5; ++i) {
        drones[i] = thread(&Drone::despegar, &drone_o[i], ref(print),
            ref(zonas), start_time);  //inicializamos cada thread con las
            respectivas referencias y parametros
    }

    for (auto& t : drones) t.join();     //Se espera a que todos los threads
        terminen
}
```

Salida por consola

```
[T: 0ms] Drone 0 esperando para despegar...
[T: 0ms] Drone 0 despegando...
[T: 0ms] Drone 1 esperando para despegar...
[T: 0ms] Drone 2 esperando para despegar...
[T: 0ms] Drone 2 despegando...
[T: 0ms] Drone 3 esperando para despegar...
[T: 0ms] Drone 4 esperando para despegar...
[T: 5000ms] Drone 0 alcanzó altura de 10m
[T: 5000ms] Drone 4 despegando...
[T: 5000ms] Drone 2 alcanzó altura de 10m
[T: 5000ms] Drone 1 despegando...
[T: 10000ms] Drone 4 alcanzó altura de 10m
[T: 10000ms] Drone 3 despegando...
[T: 10000ms] Drone 1 alcanzó altura de 10m
[T: 15000ms] Drone 3 alcanzó altura de 10m
```

Nótese que no hay lineas de salida entremezcladas y efectivamente se espera 5 segundos por dron como es solicitado. El escenario de mejor utilización de recursos permite, por cuestiones físicas del espacio, el despegue simultaneo de 2 drones que no sean vecinos. Luego, con 15 segundos todos los drones se encuentran a una altura de 10 m.

5. Ejercicio 3. Sistema de Monitoreo y Procesamiento de Robots Autónomos

5.1. Objetivo y Descripción

En este ejercicio se desarrollará un sistema de monitoreo y procesamiento de tareas en un entorno industrial automatizado, donde múltiples sensores generan reportes de inspección que deben ser procesados por robots autónomos. El objetivo es implementar un modelo concurrente que simule esta interacción entre sensores y robots, garantizando sincronización, exclusión mutua y procesamiento ordenado.

Cada sensor actuará como un hilo independiente que produce tareas con una demora fija de 175 milisegundos por tarea. Estas tareas deben ser enviadas a una cola centralizada de tareas compartida, implementada como `std::queue<Tarea>`. Cada tarea estará representada mediante un `struct` que incluirá: un entero `idSensor` que identifica al sensor emisor, un entero `idTarea` para distinguirla, y una `std::string descripcionTarea` que incluya una descripción genérica con el número de tarea.

Los robots autónomos también serán representados por hilos, y consumirán tareas de la cola respetando el orden de llegada. El procesamiento de cada tarea por parte de un robot deberá tomar exactamente 250 milisegundos. Cuando la cola esté vacía, los robots deberán esperar de forma bloqueante hasta que haya nuevas tareas disponibles. Además, una vez que todos los sensores hayan finalizado la generación de tareas, los robots deberán procesar las tareas restantes y finalizar su ejecución correctamente.

El sistema deberá utilizar mecanismos de sincronización de la biblioteca estándar `std::mutex`, `std::lock_guard`, `std::unique_lock` y `std::condition_variable`, para garantizar condiciones seguras de concurrencia y evitar condiciones de carrera. No se permite el uso de otros mecanismos de sincronización adicionales.

La implementación incluirá un caso de prueba con tres sensores y tres robots, y deberá mostrar por consola los eventos de generación y procesamiento de tareas. Se debe garantizar que los mensajes en consola no se superpongan y que los únicos tiempos de espera utilizados sean los estipulados: 175 ms para la generación de tareas y 250 ms para su procesamiento.

5.2. Desarrollo

Estructura de clases

El desarrollo de la solución para la consigna parte de buscar comportamientos específicos de estructuras específicas del programa. Así es como se implementaron las clases Sensor, Robot, TaskQueue y Manager que junto con el struct Tarea, permiten manejar todos los eventos del entorno, interactuar sinergicámente para producir tareas y procesarlas así como encapsular la lógica del request y insert a una cola de tareas. Lo primero es la definición de la estrucutra de cada tarea, simple pero elemental, luego procedemos por las estructuras más complejas que sí encapsulan un comportamiento propio.

```
struct Tarea {
    Tarea() = default;  // Constructor para inicialización default
    Tarea(int id_sensor, int id_task, string d) : //Inicialización común
            idSensor(id_sensor), idTarea(id_task), descripcionTarea(d) {}

    int idSensor, idTarea;
    string descripcionTarea;
    // Devuelve un string con toda la info de la tarea
    string toString() const {
        return ("Tarea #" + to_string(idTarea) + " creada por: SENSOR[" +
           to_string(idSensor) + "]\nDescripción:" + descripcionTarea);
    }
};
```

Listing 2: Definicion de Tarea

- `TaskQueue`: Es un contenedor para la cola de tareas, cuenta con un mutex de acceso a la cola. Implementa una condition_variable para permitir su uso con multithreading y contadores atomicos de tareas procesadas y generadas. Además tiene una flag de estado booleano para cuando los productores de la cola han dejado de trabajar se avise (mediante un manager del entorno) a los consumidores. Lo que nos permite esta implementación es encapsular el comportamiento de una cola multithreading para independzarlo de sus trabajadores. Entonces si se quiere producir una tarea se invoca al método newTask con los parametros de construcción de la tarea y la estructura la produce e inserta en la queue. Lo mismo si se quiere consumir, se invoca popTask y se recibe.

```
class TaskQueue {
    queue<Tarea> tasks;          // Cola de tareas
    mutex handler;               // MTX de la cola
    condition_variable cv;       // Condicion variable para avisar a los
        consumidores de tareas
    atomic_bool done{false};     // Estado de produccion de tareas
        terminada {false (activa), true (finalizada)}
    atomic_int t_generated{0};   // Contador total de tareas generadas
    atomic_int t_procesed{0};    // Contador total de tareas procesadas

    public:
        TaskQueue() = default;
        // Permite insert de una tarea dado el id y la descripcion
        void newTask(int sensor, string descripcion);
        // Hace pop de una tarea si la cola se encuentra disponible,
            devuelve true si sucede y escribe el objeto sobre la
            referencia pasada
        bool popTask(Tarea& task);
        // Cambia el estado de done a true
        void setDone();
        // Devuelve el estado de done
        bool Done() const {return done.load();}
        // Devuelve la cantidad de tareas generadas
        int getGen() const {return t_generated.load();}
        // Devuelve la cantidad de tareas procesadas
        int getPro() const {return t_procesed.load();}
        ~TaskQueue() = default;
};
```

Listing 3: Definicion de TaskQueue

- `Sensor`: Es un sensor que genera tareas de forma independiente en sinergía con el Manager y la TaskQueue. Cuenta con un thread que trabaja de forma paralela y es lo que permite hacer una gran cantidad de sensores trabajando en simultáneo para sensar y generar tareas en base a las mediciones.

```
class Sensor {
    TaskQueue &queue;    // Cola de tareas para la cual produce tareas
    Manager& manager;    // Manager para el cual trabaja
    int id;              // id del Sensor
    thread worker;       // thread asignado al Sensor
    public:
        /**
         * Constructor de Sensor
         *
         * params: q (es la cola de tareas que se le asigna) m (es el
            manager que se le asigna) i (el id del sensor) n_tasks (la
            cantidad de tareas que va a producir)
         */
        Sensor(TaskQueue& q, Manager& m, int i, int n_tasks) :
```

```
            queue(q), manager(m), id(i) {
                worker = thread([this, n_tasks](){produce(n_tasks); });
            }

        // Función de producción de tareas. Recibe la cantidad de
           tareas que el Sensor tiene que generar
        void produce(int n_tasks);

        // Destructor especificado para multithreading
        ~Sensor() {
            worker.join();  // Hacemos join para evitar que el Sensor
               sea destruido cuando su worker siga en producción
        }
};
```

Listing 4: Definicion de Sensor

- `Robot`: Es un robot que procesa tareas de forma independiente. Cuenta con un hilo que permite el uso de multiples robots trabajando en paralelo para procesar tareas de la cola que tiene asignada.

```
class Robot {
    TaskQueue &queue;    // Cola de tareas asignada
    int id;              // id del Robot
    thread worker;       // thread del Robot
    public:
        /**
         * Constructor de Robot
         *
         * params: q (es la cola de tareas que se le asigna) i (el id
            del Robot)
         *
         * idea:  Es analogo a Sensor solo que implemetado para el
            worker de Robot, poniendolo a consumir e inicializando sus
            miembros, cola e id
         */
        Robot(TaskQueue& q, int i) :
            queue(q), id(i) {
                worker = thread([this] { consume(); });
            }

        // Funcioin de consumo de taeras
        void consume();

        // Destructor especificado para multithreading
        ~Robot() {
            worker.join();  // Hacemos join para evitar que el Robot
               sea destruido cuando su worker siga en produccion
        }
};
```

Listing 5: Definicion de Robot

- `Manager`: Es un Manager de eventos. Basicamente es quien interactua con la cola, los sensores y los robots. Es el encargado de asginar la cola a sensores y robots, cambiar el estado de prodcucción de la cola y avisar a los sensores cuando la cola deja de estar en funcionamiento Cuenta con contenedores de Sensor y Robot implementado con listas (para evitar move de la estructura por extension) y así implementar o simular el funcionamiento en paralelo de los sensores y robots junto con la cola.

```
class Manager {
```

```
    TaskQueue& queue;                       // Cola de tareas con la cual trabaja
    list<Sensor> sensores;                  // lista de sensores a cargo
    list<Robot> robots;                     // lista de robots a cargo
    atomic_int sensoresTerminados{0};       // contador de sensores terminados

    public:
        /**
         * Constructor de Managaer
         *
         * params: q (cola de tareas a asignar) sensoresCount (cantidad de sensores a cargo) robotsCount (cantidad de robots a cargo) n_tasks (la cantidad de tareas que cada sensor debe producir)
         *
         * idea: Se asgina la cola, luego en el cuerpo se inicializan las listas de sensores y de robots. Se hace uso de emplace ya que llama al constructor respectivo del contenido pasandole los parametros de interes.  Notar que cuando es asignado, cada robot/sensor arranca su tarea
         */
        Manager(TaskQueue& q, int sensoresCount, int robotsCount, int n_tasks) : queue(q)
        {
            for (int i = 0; i < sensoresCount; ++i)
                sensores.emplace_back(queue, *this, i, n_tasks);    // Se le pasa this para que el sensor asgine al manager que lo construye
            for (int i = 0; i < robotsCount; ++i)
                robots.emplace_back(queue, i);
        }

        // Verifica si la cola finalizo su produccion al terminar un sensor y avisa a los robots de ser asi
        void notifySensorDone() {if (++sensoresTerminados == int(sensores.size())) queue.setDone();}    // Si los terminados son igual a los totales, seteamos la cola finalizada

        ~Manager() = default;
};
```

Listing 6: Definicion de Manager

Lógica de TaskQueue

Para aislar la funcionalidad de la cola independizando de los agentes que contribuyen su comportamiento se implementaron los métodos cruciales que fueron presentados en la definición (Listing 3). Vamos a desarrollar los fundamentales de inserción y extracción de tareas. Los demás son cuestión de revisión de estados de flags y contadores atómicos con lógica elemental explicada en el código.

- `void TaskQueue::newTask(int sensor, string descripcion)`: Este método permite hacer insert de tareas, la lógica es simple, se intenta ganar el mutex de manejo de la cola, luego se hace emplace con los parámetros (en ese momento se aumenta el contador de tareas generadas) de la tarea y se avisa a los consumidores esperando por la condition_variable.

```
void TaskQueue::newTask(int sensor, string descripcion) {
    {
    lock_guard<mutex> lock(handler); // Hacemos lock_guard para evitar unlock, cuando salga de scope se desbloquea automaticamente
```

```
    tasks.emplace(sensor, t_generated++, descripcion);  // Inserta una
        tarea nueva en la cola con los parametros correspondientes
    }
    cv.notify_one();     // IMPORATANTE: cuando se genera una tarea se
        avisa a un consumidor que hay una tarea disponible mediante la
        condition_vairable
    // Con esto garantizamos que si un consumidor estaba en espera, se
        despiere su worker y verifique si puede consumir y efectivamente
        se ponga a consumir
}
```

- `bool TaskQueue::popTask(Tarea& task)`: El pop encapsula un poco más de lógica. Primero se accede al handler pero por unique_lock para poder transferir el mutex. Luego se invoca wait sobre la cv, pasándole el lock y una lambda que evalúa si la cola está vacía o fuera de producción. Notemos que por la esencia del wait, si el predicado es verdadero no pone en espera al thread. Entonces, solo entra en espera cuando la cola está vacía y sigue la producción. En esa situación el thread esperará a que la cola tenga una nueva tarea para continuar.

  Luego viene el if-statement que verifica si la cola se encuentra vacía. Si es así se retorna false ya que no se pudo hacer pop. Sin embargo, si hay tareas en la cola, significa que independientemente al estado de producción de la cola todavía hay tareas por procesar, por lo que procede a acceder a la próxima tarea, eliminarla de la queue y aumentar el contador de tareas procesadas retornando true al final.

```
bool TaskQueue::popTask(Tarea& task) {
    unique_lock<mutex> lock(handler);   // unique_lock para poder hacer
        uso de cv.wait

    // Se pone en espera de la condition_variable. Entonces se
        trnasfiere el lock y se pasa la lambda
    // que verificará al despertarse el thread y antes de ponerse en
        espera.
    cv.wait(lock, [this]() { return !tasks.empty() || done.load(); });
    // Es importante que cuando salga de wait, se verifique de nuevo
        que haya tareas por eso no es else if
    if (!tasks.empty()) {   // Veriffica si hay tareas disponibles
        task = tasks.front();   // Guarda la tarea próxima en la
            referencia del parametro
        tasks.pop();    // Popea la tarea próxima
        t_procesed ++;  // Aumenta el contador de tareas procesadas
        return true;    // Devuelve el aviso
    }
    // false se devuelve únicamente cuando efectivamente la cola se
        encuentra vacia y fuera de produccion (finalizada)
    else return false;
}
```

Lógica de Sensor y Robot

Dentro de esta subsección vamos a explicar el comportamiento de los sensores y robots los cuales presentan ideas generales compartidas ya que ambos trabajan con threads. Para agregar una descripción del comportamiento de cada Sensor/Robot (Listings 4 y 5), queremos explicar la lógica detrás del constructor de la clase y de los métodos produce/consume.

```
Sensor::Sensor(TaskQueue& q, Manager& m, int i, int n_tasks) :
    queue(q), manager(m), id(i) {
        worker = thread([this, n_tasks](){produce(n_tasks); });
    }

```

```
    Robot::Robot(TaskQueue& q, int i) :
        queue(q), id(i) {
            worker = thread([this] { consume(); });
        }
```

Hay una particularidad en el constructor del sensor y es la forma de inicializar sus miembros. Luego de la lista de inicialización convencional que siempre utilizamos se le asigna al worker un objeto thread cuya función es una lambda. Primero, notar que worker es reasignado por el `"="`, esto no es un problema ya que la inicialización default del thread no lo bindea a una función. Segundo, la función lambda captura del scope el Sensor creado mediante `"this"` y el numero de tareas con el que el sensor fue asignado. No recibe parámetros y lo que hace en su cuerpo es llamar a la función `produce(int)` que pone a trabajar al worker instantaneamente. Con esto lo que garantizamos es que el worker sea inicializado correctamente con su función respectiva y que apenas se inicializa el Sensor se lo ponga a producir tareas. Otra forma de inicializarlo sería en la lista mediante: `worker([this, n_tasks](){produce(n_tasks); })` El problema que puede surgir es que la lista no garantiza una inicialización específica. Esto significa que podríamos inicializar el miembro worker sin que se haya inicializado antes la cola o el manager, lo cual sí implica un problema al tener que interactuar con ellos. El constructor de Robot es análogo a Sensor solo que implementado para el worker de Robot, poniéndolo a consumir e inicializando sus miembros, cola e id.

Con respecto a los métodos produce y consume denotados a continuación, encapsulan el comportamiento respectivo de extraer/insertar en la cola con una espera temporal que emula el tiempo de trabajo especificado en la consigna. La lógica de cada una es sencilla y esto es gracias a que el comportamiento de la cola fue manejado de forma aislada en dicha estructura. Quien eventualmente interactúa con otro agente del entorno es Sensor (notifica a Manager que terminói su producción)

```
void Sensor::produce(int n_tasks) {
    for (int i = 0; i < n_tasks; ++i) { //Arranca el bucle para la cantidad
        de tareas que tiene que producir
        this_thread::sleep_for(chrono::milliseconds(175));  // Frena 175 ms
            simulando el tiempo de producción
        queue.newTask(id, "Esta es una tarea de prueba");   // Inserta la
            tarea generada en la cola
        safe_print("Sensor #" + to_string(id) + " generó una tarea");   //
            Avisa que genero una tarea (uso depurativo)
    }
    manager.notifySensorDone(); // IMPORTANTE: cuando termina de producir
        avisa al manager que el sensor salió de producción
}

void Robot::consume() {
    Tarea task; // Es la tarea que se va a ir popeando cuando consuma de la
        cola
    while (queue.popTask(task)) {   // Mientras se popee de la cola, el
        resultado será true, task tendrá una nueva tarea para procesar
        safe_print("Robot #" + to_string(id) + " procesando..."); // Avisa
            que está procesando
        this_thread::sleep_for(chrono::milliseconds(250));  // Espera 250
            ms simulando el tiempo de procesamiento
        safe_print("Robot #" + to_string(id) + " procesó:\n"+
            task.toString()); // Avisa lo que procesó
    }
    // Notemos que saldrá del loop unicamente cuando (por diseño de la
        cola) la cola haya salido de producción

    safe_print("Robot #" + to_string(id) + " no procesa más"); // Avisa
        cuando sale de procesamiento
}
```

Finalmente, cabe destacar el ajuste al destructor de cada una de las clases, el mismo incluye

`worker.join()` para evitar que el objeto sea destruido aun siendo que el thread se encuentra en pleno trabajo. Entonces el destructor joinea al thread y una vez que este termina se termina de destruir el objeto.

Lógica de Manager

Como manejador de eventos Manager tiene a cargo un cola de tareas la cual tiene que garantizar su producción de tareas y procesamiento de las mismas de forma eficiente. Para ello implementamos la estructura (Listing 6). La idea de esta estructura es que controle el lifetime de los robots y sensores, viven en el gracias a que el contenedor de estos esta almacenado en la estructura.

La implmentación de los contenedores viene de la mano de std::list para garantizar que no se mueva la estructura por realloc si crece en tamaño. Esto es un problema para los consumidores y productores ya que hacer realloc podría interrumpir la tarea del thread al estar trabajando y tener que realocar los objetos construyendo nuevos. List vive en stack al ser nodos enlazados doblemente, quizá el detalle del doble enlazado sea redundante y utilice mayores recursos, en una futura atualización de la utilidad de la estructura se podría buscar entre los hilos, ordenarlos según su eficiencia o alguna otra funcionalidad que aproveche el doble enlace.

La inicialización de los sensores y robots es secuencial, esto quiere decir que el bucle for es quien se encarga de inicializar todos los sensores primero y todos los robots luego (según la cantidad de sensores y robots que se le indique al manager). A pesar de esto los workers actúan en simultáneo gracias a la implementación del multihilo.

Finalmente, Manager es quien se encarga de chequear si todos los sensores salieron de producción haciendo uso de `void notifySensorDone()`. Esta función permite a Sensor (quien interactúa directamente con Manager, avisarle que finalizó su producción, Manager aumenta el cotnador interno y si resulta ser igual a la cantidad de sensores totales, cambia el estado de la condition_variable de la queue gracias al método de la cola `void TaskQueue::setDone()`.

Utilidades

Un detalle del manejo de todo el entorno, hay dos variables gloabales relevantes, por un lado el contador de inicio del programa declarado en `events.hpp` e inicializado en `events.cpp` (donde es utilizado). Las herramientas de este timestamp son provenietnes de la librería `std::chrono`. Por otro lado el mutex del cout, también declarado extern en el header y definido en la implementación. Son utilidades que se usan en la función de `safe_print(string)` la cual permite printear un mensaje por consola haciendo uso de un recurso compartido (el cout) evitando race conditions.

```
1  const chrono::steady_clock::time_point START_T =
       chrono::steady_clock::now(); //Inicializamos el cronometro global del
       inicio del programa
2  mutex COUT_MTX; //Inicializamos el mutex
3
4  void safe_print(string message) {
5      auto now = chrono::steady_clock::now(); // Toma el tiempo exacto en el
           que se invoca el print
6      auto elapsed = chrono::duration_cast<chrono::milliseconds>(now -
           START_T).count();  // Calcula el tiempo transcurrido
7      COUT_MTX.lock();     // Lockea el mutex, si no puede se queda en espera
8      cout << "[T: " << elapsed << "ms] " << message << endl; // Una vez que
           tiene el mutex usa cout (se imprime el tiempo y el mensaje)
9      COUT_MTX.unlock();  // Desbloque el mutex
10 }
```

Implementación en `main.cpp`

Al encapsular todos los comportamientos en estructuras interdependientes de main, el código dentro de esta sección es bastante limpio.

```cpp
int main(){
    safe_print("Inicio del programa");  // Para garantizar la seguridad del cout
    TaskQueue q;         // Inicializamos la cola de tareas
    {
        Manager(q, 3, 3, 1); // Se inicializa el manager con 3 sensores y 3 robots. Cada sensor genera 1 tarea
    }
    // A partir de este punto los threads están destruidos por lo que cout es seguro de usar
    cout << "Los eventos de creacion y procesamiento de tareas han finalizado"
            << "\nTOTAL GENERADAS: " << q.getGen()
            << "\nTOTAL PROCESADAS: " << q.getPro()
            << endl;
}
```

Lo único importante es notar que la cola se crea por fuera del Manager (su lifetime es independiente) y el scope del Manager emula una definicóin específica con una salida del entorno inmediata aunque no destructiva (Manager al ser un objeto del scope es destruido pero antes deben destruirse sus estructuras contenidas, osea las listas de sensores y robots, recordemos que cada uno se destruirá cuando el worker finalice su tarea). Por esto, es seguro que cuando se haya destruido Manager la cola estará completamente vacía y fuera de producción, ningún thread seguirá corriendo y entonces el cout es seguro de usarse libremente. Se imprimen los datos registrados sobre las tareas a modo de evaluación de la funcionalidad de la implementación.

Ejemplo de salida por consola

Como bien es sabido el uso de threads no es determinístico, con lo cual, el sensor 2 podría genera tareas antes que el sensor 0 de la misma forma que el robot podría terminar de procesar tareas antes que el robot 1. Sin embargo, notemos la persistencia temporal de los timestamps del ejemplo de ejecución donde se puede ver que a los 175 ms aprox. se generan las tareas y a los 425 ms aprox (250 ms después) las tares son procesadas. Nótese el aviso de los robots al procesar y terminar.

```
[T: 0ms] Inicio del programa
[T: 175ms] Sensor #0 generó una tarea
[T: 175ms] Sensor #1 generó una tarea
[T: 175ms] Robot #0 procesando...
[T: 175ms] Robot #1 procesando...
[T: 175ms] Sensor #2 generó una tarea
[T: 175ms] Robot #2 procesando...
[T: 425ms] Robot #0 procesó:
Tarea #0 creada por: SENSOR[0]
Descripción:Esta es una tarea de prueba
[T: 425ms] Robot #0 no procesa más
[T: 425ms] Robot #1 procesó:
Tarea #1 creada por: SENSOR[1]
Descripción:Esta es una tarea de prueba
[T: 425ms] Robot #1 no procesa más
[T: 425ms] Robot #2 procesó:
Tarea #2 creada por: SENSOR[2]
Descripción:Esta es una tarea de prueba
[T: 425ms] Robot #2 no procesa más
Los eventos de creacion y procesamiento de tareas han finalizado
TOTAL GENERADAS: 3
TOTAL PROCESADAS: 3
```

6. Conclusiones

El desarrollo del Trabajo Práctico 2 nos permitió aplicar de manera práctica múltiples conceptos fundamentales de la programación en C++ y la forma de utilizar los diferentes paradigmas de la programación previstos en la materia con un fin específico aun así complementario. Nos permitió trabajar con herramientas como la gestión de contenedores STL, la sobrecarga de operadores, la serialización binaria, y sobre todo, la programación concurrente con sincronización de hilos.

En el primer ejercicio, la principal dificultad surgió del uso correcto de `std::unordered_map` con claves personalizadas, lo cual nos obligó a implementar operadores de igualdad y especializaciones de `std::hash`, cuidando también la organización de los archivos para evitar errores de compilación. La extensión del sistema con persistencia binaria implicó un manejo cuidadoso de la lectura y escritura de estructuras complejas, validando archivos corruptos o vacíos para evitar efectos indeseados.

En el segundo ejercicio, el diseño de una solución multihilo para el sistema de drones supuso el desafío de evitar *deadlocks* en la adquisición simultánea de recursos compartidos. El uso correcto de `std::lock` junto con `std::mutex` y el diseño de un sistema de log seguro para consola fueron claves para garantizar una simulación realista y libre de condiciones de carrera.

En el tercer ejercicio, nos enfrentamos a una situación más compleja donde múltiples productores (sensores) y consumidores (robots) acceden concurrentemente a una cola compartida. El diseño e implementación de una `TaskQueue` con `std::condition_variable` nos permitió sincronizar eficientemente el acceso a tareas, asegurando que todos los hilos trabajen sin bloqueos innecesarios ni superposición de mensajes en consola. El resultado de todo esto fue una implementación minuciosa de la programación orientada a objetos en conjunto con la concurrente que obsequió un entendimiento más detallado de como los sistemas y programas son diseñados. Por demás está mencionar que nos abre la posibilidad a pensar en las aplicaciones prácticas por fuera del marco del trabajo práctico y la materia, implementación en servidores y sistema de gestión de flujo de clientes, etc.

En conjunto, este trabajo práctico consolidó nuestro entendimiento sobre la programación orientada a objetos, la manipulación segura de estructuras de datos y la programación concurrente en C++. Quedan abiertas líneas de mejora como la ampliación del número de agentes en los sistemas simulados, el agregado de logs persistentes y la validación de cargas parciales o fusionadas en la Pokedex.