Reporte Técnico del Sistema de Gestión de Sensores

Introducción

El presente sistema es una simulación de un sistema de gestión de lecturas de sensores (IoT) implementado en C++. Su objetivo principal es demostrar la aplicación de estructuras de datos genéricas (listas enlazadas simples) y el polimorfismo en la gestión unificada de diferentes tipos de sensores que manejan distintos tipos de datos (como float para temperatura e int para presión). El sistema se ha diseñado sin utilizar la Standard Template Library (STL), gestionando la memoria de forma dinámica con punteros.

Manual Técnico

2.1. Diseño y Arquitectura

La arquitectura del sistema se basa en la herencia y el polimorfismo para crear una jerarquía de clases que permite manejar de forma uniforme a los diferentes tipos de sensores.

Componentes Clave:

- Clase ListaSensor<T>: Es una clase plantilla genérica que implementa una lista enlazada simple para almacenar datos del tipo T.

Propósito: Almacenar el historial de lecturas de un sensor específico.
Características: Manejo dinámico de memoria, soporta cualquier tipo de dato (T), e incluye métodos básicos de lista (push_back, buscar, removeValue) y funciones de procesamiento (average, removeLowest).

- Clase Abstracta SensorBase: Define la interfaz común para todos los sensores del sistema.

Propósito: Proveer una base polimórfica para la gestión unificada.
Miembros: Incluye el identificador (nombre) del sensor y los métodos virtuales puros procesarLectura() e imprimirInfo().

- Clases Concretas de Sensores (SensorTemperatura, SensorPresion): Heredan de SensorBase.

- SensorTemperatura: Maneja lecturas de tipo float y utiliza ListaSensor<float>. Su método procesarLectura() calcula el promedio y elimina la lectura más baja (removeLowest).

- SensorPresion: Maneja lecturas de tipo int y utiliza ListaSensor<int>. Su método procesarLectura() simplemente calcula y muestra el promedio.

- Clase ListaGeneral: Implementa una lista enlazada simple específica para gestionar punteros a SensorBase.

Propósito: Actúa como el contenedor principal del sistema, permitiendo almacenar y recorrer todos los sensores (temperatura y presión) de forma polimórfica.
Funcionalidad: Permite insertar nuevos sensores, buscarPorID y ejecutar procesarTodos() e imprimirTodos(), delegando la acción a los objetos específicos de cada sensor a través del polimorfismo.

Desarrollo

El desarrollo se enfoca en tres áreas principales: la gestión de la lista genérica, el diseño de la herencia y el manejo de la memoria.

- Gestión de Lista Genérica (ListaSensor.h):

RAII y Deep Copy: Se implementan el Constructor de Copia y el Operador de Asignación con lógica de copia profunda (Deep Copy). Esto asegura que la lista de destino no comparta punteros con la lista origen, previniendo errores de doble liberación de memoria.

- Gestión de Memoria: El Destructor (~ListaSensor()) y el método clear() se encargan de liberar explícitamente la memoria de cada Nodo creado dinámicamente.

Operaciones:
push_back(): Añade un nuevo nodo al final.
average(): Calcula el promedio, realizando un static_cast<double> para asegurar la precisión del resultado.

removeLowest(): Recorre la lista para encontrar el nodo con el valor más bajo y lo elimina, reajustando los punteros.

- Implementación Polimórfica (main.cpp):

Herencia: Las clases SensorTemperatura y SensorPresion heredan de SensorBase y anulan sus métodos virtuales puros.

Gestión Centralizada: La clase ListaGeneral es el núcleo de la gestión. Al recorrer su lista de punteros a SensorBase y llamar a métodos como procesarLectura(), se invoca la implementación correcta en tiempo de ejecución (ej. la de SensorTemperatura o SensorPresion) gracias a las funciones virtuales.

Downcasting: El programa principal (main) utiliza dynamic_cast para intentar convertir un puntero SensorBase* a su tipo derivado específico (SensorTemperatura* o SensorPresion*) al momento de registrarLectura(), lo que permite llamar al método específico de registro de lectura.

Control de Memoria en la ListaGeneral:
El destructor de ListaGeneral y su método clear() son cruciales. No solo eliminan los nodos de la lista (delete aux), sino que también se encargan de eliminar el objeto SensorBase al que apunta cada nodo (delete aux->sensor), previniendo fugas de memoria y asegurando que se llame al destructor virtual de las clases derivadas.