Reporte Técnico: Sistema de Gestión Polimórfica de Sensores para IoT

David Alejandro Ibarra Castañeda

31 de octubre de 2025

Índice

1. Introducción

El presente documento detalla el diseño y desarrollo de un sistema de bajo nivel para una empresa de monitoreo de Infraestructura Crítica (IC). El objetivo principal es registrar, almacenar y procesar lecturas de múltiples tipos de sensores (como temperatura, presión, etc.) de manera unificada y flexible.

El sistema fue diseñado para superar dos limitaciones de diseño fundamentales:

- **Rigidez del Tipo de Dato:** Las lecturas pueden ser de distintos tipos (enteros, punto flotante). El sistema debía manejar esta genericidad sin duplicar código.
- **Rigidez de la Estructura:** El número de lecturas es variable y desconocido en tiempo de compilación, exigiendo una estructura de datos dinámica.

La solución implementada se basa en una **Jerarquía de Clases Polimórfica** que gestiona **Listas Enlazadas Simples Genéricas** (‘ListaSensor¡T¿‘). Este diseño permite que cualquier tipo de sensor (como subclase) sea agregado a una única lista de gestión, forzando la implementación de métodos esenciales a través de Clases Abstractas.

El proyecto cumple con requisitos no funcionales estrictos, como la prohibición de la STL (Standard Template Library), obligando a una gestión de memoria manual mediante punteros, y la implementación de la Regla de los Tres”para evitar fugas de memoria.

2. Manual Técnico

2.1. Diseño Arquitectónico

El diseño del sistema se fundamenta en tres pilares clave de la Programación Orientada a Objetos (POO) avanzada y la gestión de memoria en C++.

2.1.1. Pilar 1: Polimorfismo y Clases Abstractas

El núcleo de la flexibilidad del sistema reside en el polimorfismo. Se definió una clase base abstracta ‘SensorBase‘ que actúa como una ïnterfaz.º contrato.

```
class SensorBase {
protected:
    char nombre[50];
public:
    virtual ~SensorBase(); // Destructor virtual

    // Metodos virtuales puros
    virtual void procesarLectura() = 0;
    virtual void imprimirInfo() const = 0;
    virtual void registrarNuevaLectura(Serial& port) = 0;

    const char* getNombre() const;
};
```

Listing 1: Interfaz de la Clase Base Abstracta (Sensor.h)

Cualquier sensor concreto, como ‘SensorTemperatura‘ o ‘SensorPresion‘, debe heredar de ‘SensorBase‘ e implementar estos métodos.

La clase ‘Sistema‘ (el gestor principal) aprovecha esto manteniendo una única lista de punteros a la base: ‘ListaSensor¡SensorBase*¿‘. Esto le permite almacenar diferentes tipos de sensores en

una misma estructura y ejecutar ‘procesarLectura()‘ sobre ellos sin saber de qué tipo específico son.

2.1.2. Pilar 2: Genericidad (Templates)

Para resolver la Rigidez del Tipo de Dato”, se implementó una clase genérica ‘ListaSensor¡T¿‘ usando plantillas (templates). Esta única clase es capaz de manejar las listas enlazadas para cualquier tipo de dato, logrando una reutilización de código completa.

En el proyecto, ‘ListaSensor¡T¿‘ se instancia de tres formas distintas:

- ‘ListaSensor¡float¿‘: Usada dentro de ‘SensorTemperatura‘ para almacenar su historial.
- ‘ListaSensor¡int¿‘: Usada dentro de ‘SensorPresion‘ para almacenar su historial.
- ‘ListaSensor¡SensorBase*¿‘: Usada por la clase ‘Sistema‘ como la lista de gestión polimórfica.

2.1.3. Pilar 3: Gestión de Memoria Manual

Dado el requisito de no usar STL, la gestión de memoria es explícita.

- **Regla de los Tres:** La clase ‘ListaSensor¡T¿‘ implementa un destructor, un constructor de copia y un operador de asignación. Esto es vital para asegurar que al copiar o destruir una lista, se realice una çopia profunda”(deep copy) de los nodos y se libere la memoria correctamente, evitando fugas o punteros corruptos (dangling pointers).
- **Destructor Virtual y Liberación en Cascada:** La clase ‘SensorBase‘ define su destructor como ‘virtual‘. Esto es el mecanismo más crítico para la correcta liberación de memoria. Cuando el ‘Sistema‘ se destruye, itera sobre su lista ‘ListaSensor¡SensorBase*¿‘ y llama a ‘delete‘ sobre cada puntero. Gracias al destructor virtual, el sistema invoca al destructor de la clase derivada correcta (ej. ‘ SensorTemperatura()‘), el cual a su vez invoca al destructor de su ‘ListaSensor¡float¿‘ interna, liberando así toda la memoria en una reacción en cascada.

2.2. Desarrollo y Compilación

2.2.1. Sistema de Compilación (CMake)

El proyecto se gestiona mediante CMake. El archivo ‘CMakeLists.txt‘ define el ejecutable (‘monitor‘) y enlaza todas las implementaciones (‘.cpp‘) necesarias.

```
cmake_minimum_required(VERSION 3.10)
project(MonitorIC CXX)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

add_executable(monitor
    main.cpp
    Serial.cpp
    Sensor.cpp
    Sistema.cpp
)

# Enlaza la libreria 'pthread' necesaria para
# la comunicacion serial en Linux
```

```
16 target_link_libraries(monitor PRIVATE pthread)
```

Listing 2: Archivo CMakeLists.txt

2.2.2. Comunicación Serial (Arduino)

La simulación de datos se realiza con un dispositivo Arduino (ej. Arduino UNO).

- **Emisor (Arduino):** El sketch 'SensorSim.ino' se carga en el Arduino. Este programa simula lecturas y las envía por el puerto serial usando prefijos (ej. "T:45.3"para temperatura, "P:80"para presión).
- **Receptor (C++):** La clase 'Serial' en el programa C++ utiliza las librerías POSIX ('termios.h', 'fcntl.h', 'unistd.h') para conectarse al puerto serial (ej. '/dev/ttyACM0') a una velocidad de 9600 baudios. Lee los datos línea por línea, y los métodos 'registrarNuevaLectura' de cada sensor se encargan de filtrar los prefijos ("T:.º "P:") que les corresponden.

2.2.3. Documentación (Doxygen)

Todo el código fuente (principalmente los archivos '.h') fue documentado siguiendo los estándares de Doxygen. Se utiliza un 'Doxyfile' para configurar la generación automática de la documentación del código en formato HTML, incluyendo diagramas de herencia de clases generados por Graphviz.

2.3. Componentes del Sistema

El proyecto está modularizado en las siguientes clases y archivos:

`ListaSensor.h` Clase de lista enlazada simple genérica ('template ¡typename T¿'). Gestiona los 'Nodo¡T¿', la inserción, búsqueda y la correcta liberación de memoria (Regla de los Tres).

`Sensor.h / Sensor.cpp` Define la jerarquía polimórfica.

- **SensorBase:** Clase abstracta con métodos virtuales puros ('procesarLectura', 'registrarNuevaLectura').
- **SensorTemperatura:** Clase concreta. Hereda de 'SensorBase' y contiene una 'ListaSensor¡float¿'.
- **SensorPresion:** Clase concreta. Hereda de 'SensorBase' y contiene una 'ListaSensor¡int¿'.

`Serial.h / Serial.cpp` Clase "wrapper"que encapsula la complejidad de la comunicación con el puerto serial en Linux. Maneja la configuración ('termios') y la lectura de datos.

`Sistema.h / Sistema.cpp` Clase principal u orquestadora. Contiene la lista de gestión polimórfica ('ListaSensor¡SensorBase*¿'). Es responsable de agregar sensores, buscarlos y ejecutar el procesamiento polimórfico. Su destructor implementa la liberación en cascada.

`main.cpp` Punto de entrada del programa. Contiene el menú interactivo del usuario, inicializa el objeto 'Sistema' y el objeto 'Serial', y gestiona el bucle principal de la aplicación.

`CMakeLists.txt` Archivo de configuración de CMake para compilar el proyecto en un sistema redistribuible.

`SensorSim.ino` (Archivo externo) Sketch de Arduino utilizado como simulador de hardware para enviar datos de sensores a través del puerto USB/Serial.