UNIVERSIDAD POLITÉCNICA DE VICTORIA

30/10/25

Nombre de la Materia:

Estructura de Datos

Actividad 1 - Listas enlazadas simples

Carrera:

Ingeniería en Tecnologías de la Información e Innovación Digital.

Alumno:

Eliezer Mores Oyervides

2430037

Catedrático:

DR. SAID POLANCO MARTAGÓN

septiembre - diciembre de 2025

1. INTRODUCCIÓN

1.1 Contexto del Proyecto

En la era del Internet de las Cosas (IoT), la gestión eficiente de múltiples sensores heterogéneos representa un desafío técnico significativo. Los sistemas modernos de monitoreo requieren soluciones que permitan:

- Escalabilidad: Agregar nuevos tipos de sensores sin modificar el código base existente
- Eficiencia: Gestión óptima de recursos de memoria mediante técnicas manuales
- Flexibilidad: Procesamiento específico y diferenciado para cada tipo de sensor
- Integración: Comunicación en tiempo real con hardware físico (Arduino)
- Polimorfismo: Tratamiento uniforme de objetos heterogéneos

Este proyecto implementa un Sistema de Gestión Polimórfica que aborda estos desafíos mediante el uso de técnicas avanzadas de Programación Orientada a Objetos (POO) en C++, específicamente:

- Clases abstractas con métodos virtuales puros
- Programación genérica mediante templates
- Gestión manual de memoria dinámica
- Comunicación serial con dispositivos embebidos
- Estructuras de datos implementadas desde cero (sin STL)

1.2 Objetivo General

Desarrollar un sistema de bajo nivel para registrar, almacenar y procesar lecturas de múltiples tipos de sensores (temperatura en FLOAT, presión en INT) de manera unificada y polimórfica, implementando conceptos avanzados de POO, estructuras de datos dinámicas y comunicación con hardware embebido (Arduino).

1.3 Objetivos Específicos

1. Implementar jerarquía polimórfica usando clase base abstracta (SensorBase) y clases derivadas
2. Desarrollar dos tipos de estructuras de datos:
- Lista genérica con templates (ListaSensor<T>)
- Lista no genérica para polimorfismo (ListaGestion)
3. Aplicar programación genérica mediante templates de C++ para independencia de tipos
4. Gestionar memoria manualmente siguiendo la Regla de los Tres (constructor de copia, operador de asignación, destructor)
5. Integrar hardware real (Arduino) mediante comunicación serial POSIX
6. Garantizar modularidad y extensibilidad del sistema mediante diseño orientado a objetos
7. Procesar datos diferenciadamente:
- Temperatura: Elimina mínimo y calcula promedio
- Presión: Calcula promedio directo

1.4 Alcance del Proyecto

Funcionalidades Implementadas:

Gestión Polimórfica de Sensores - Clase base abstracta SensorBase con métodos virtuales puros - Clases derivadas: - SensorTemperatura (almacena datos FLOAT) - SensorPresion (almacena datos INT) - Procesamiento polimórfico mediante punteros a clase base - Destrucción en cascada con destructores virtuales

Dos Tipos de Estructuras de Datos - ListaSensor: Lista enlazada genérica con templates - Almacena lecturas individuales (int o float) - Implementa operaciones: insertar, calcular promedio, eliminar mínimo - Gestión manual de nodos

- ListaGestion: Lista enlazada NO genérica
- Almacena punteros a SensorBase*
- Permite polimorfismo en tiempo de ejecución
- Gestiona múltiples tipos de sensores uniformemente

Gestión de Memoria Explícita - Implementación completa de la Regla de los Tres: - Constructor de copia profunda - Operador de asignación con protección contra auto-asignación - Destructor con liberación recursiva - Sin uso de STL (std::list, std::vector, std::string) - Control total sobre asignación/liberación con new/delete

Integración con Arduino - Comunicación serial POSIX (Linux/macOS) - Lectura en tiempo real del puerto USB - Detección automática de tipo de dato (float vs int) - Configuración 9600 baud, 8N1

Procesamiento Diferenciado - Temperatura: Elimina lectura mínima, calcula promedio del resto - Presión: Calcula promedio directo de todas las lecturas

Sistema de Menú Interactivo - Creación manual de sensores - Lectura automática desde Arduino - Registro manual de lecturas - Procesamiento polimórfico - Visualización de información - Cierre ordenado con liberación de memoria

Limitaciones y Restricciones:

No se utiliza STL: Todas las estructuras implementadas manualmente Tipos limitados: Solo temperatura (float) y presión (int) Puerto serial: Compatible con Linux/macOS (requiere adaptación para Windows) Sin persistencia: Los datos no se guardan en archivos

1.5 Tecnologías Utilizadas

Tecnología	Versión	Propósito
C++	C++11	Lenguaje de programación principal
g++	4.8+	Compilador
POSIX	-	API para comunicación serial
Arduino	-	Hardware para sensores
Doxygen	1.9+	Generación de documentación
Linux	-	Sistema operativo objetivo

1.6 Metodología de Desarrollo

El proyecto siguió una metodología iterativa con las siguientes fases:

Fase 1: Diseño de Arquitectura

- Definición de la jerarquía de clases (SensorBase como raíz)
- Identificación de dos tipos de listas necesarias
- Especificación de interfaces con métodos virtuales puros

Fase 2: Implementación de Estructuras Base

- Desarrollo de ListaSensor<T> (genérica con templates)
- Desarrollo de ListaGestion (no genérica para polimorfismo)
- Implementación de la Regla de los Tres

Fase 3: Implementación de Sensores

- Clase base abstracta SensorBase
- Clases derivadas con procesamiento específico
- Pruebas de polimorfismo

Fase 4: Integración con Hardware

- Desarrollo de clase SerialPort con POSIX
- Pruebas con Arduino físico
- Detección automática de tipos

Fase 5: Documentación y Entrega

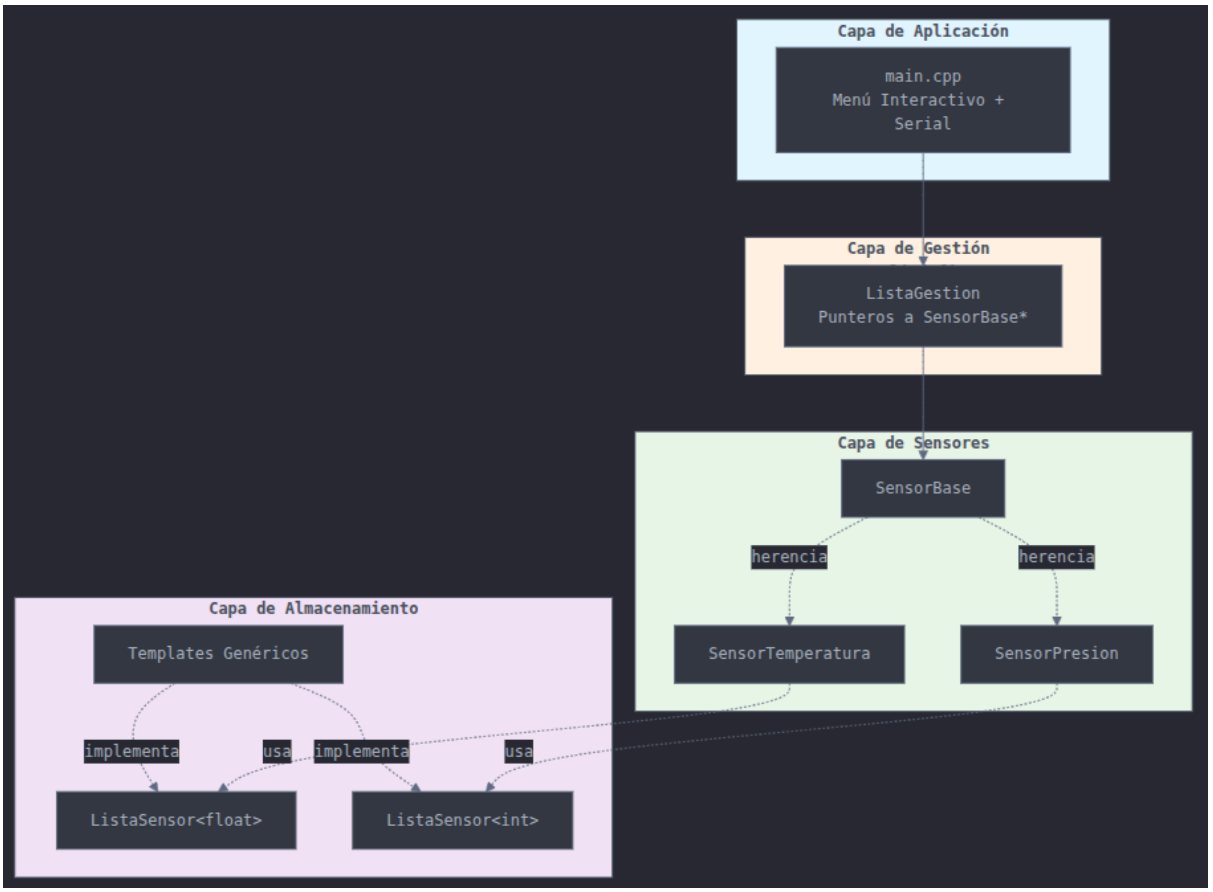
- Comentarios Doxygen en todo el código
- Generación de documentación HTML
- Creación de reporte técnico
- Verificación de requisitos

2. MANUAL TÉCNICO

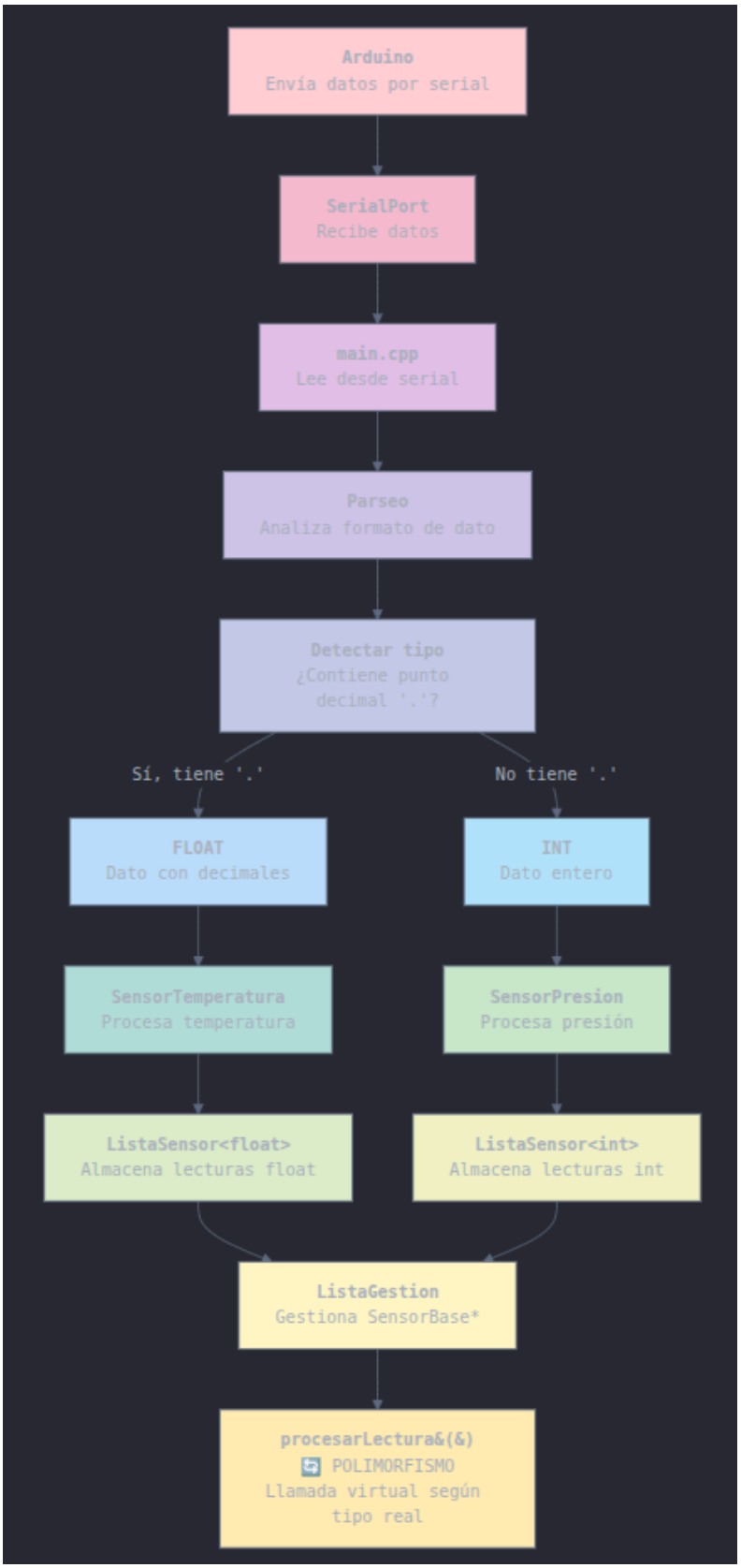
2.1 Diseño del Sistema

2.1.1 Arquitectura General

El sistema utiliza una **arquitectura en capas** con clara separación de responsabilidades:



Flujo de Datos:



2.1.2 Patrones de Diseño Implementados

1. Template Method Pattern

- Clase: SensorBase
- Propósito: Define el esqueleto de operaciones, dejando detalles a subclases
- Implementación: Métodos virtuales puros
- procesarLectura() - Cada sensor implementa su lógica
- imprimirInfo() - Cada sensor decide qué mostrar
- agregarLectura(const char*) - Cada sensor parsea según su tipo

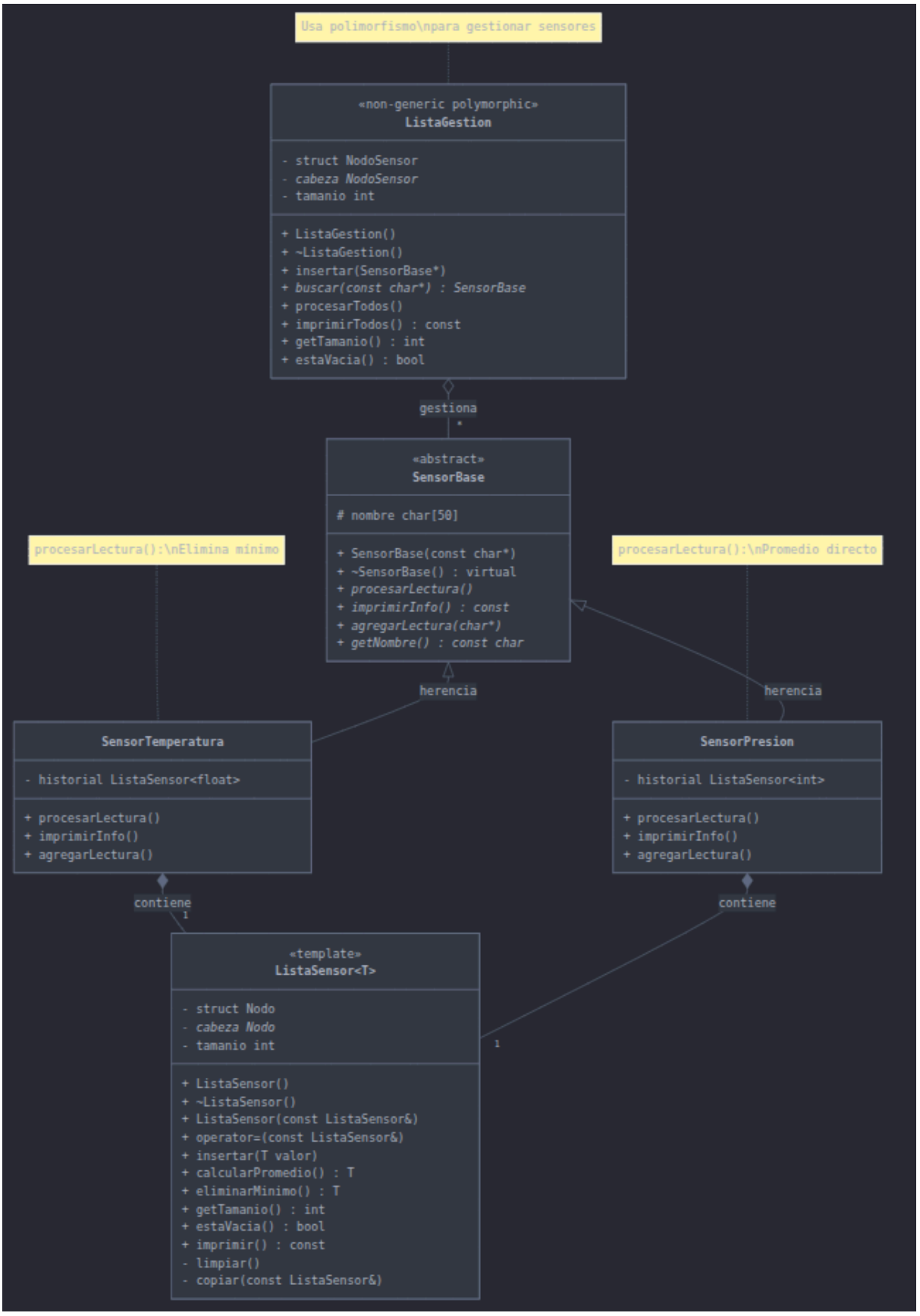
2. Strategy Pattern (Implícito)

- Diferentes estrategias de procesamiento:
- Temperatura: Elimina mínimo → Calcula promedio
- Presión: Calcula promedio directo

3. Composite Pattern (Simplificado)

- ListaGestion contiene múltiples SensorBase*
- Operación procesarTodos() itera polimórficamente

2.1.3 Diagrama de Clases UML



2.1.4 Relaciones entre Clases

Herencia: - SensorTemperatura es-un SensorBase - SensorPresion es-un SensorBase

Composición: - SensorTemperatura tiene-un ListaSensor<float> - SensorPresion tiene-un ListaSensor<int> - ListaGestion tiene-muchos SensorBase*

Agregación: - main.cpp usa ListaGestion - main.cpp usa SerialPort

2.2 Componentes

2.2.1 SensorBase.h - Clase Base Abstracta

Propósito: Define la interfaz común que todos los sensores deben implementar.

Código:

```
class SensorBase {
protected:
char nombre[50]; // NO usa std::string (requisito)

public:
SensorBase(const char* nom);
virtual ~SensorBase() {} // Destructor VIRTUAL

// Métodos virtuales PUROS (= 0)
virtual void procesarLectura() = 0;
virtual void imprimirInfo() const = 0;
virtual void agregarLectura(const char* valor) = 0;

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

Decisiones de Diseño: - char[50] en lugar de std::string (sin STL) - Destructor virtual para destrucción polimórfica correcta - Métodos virtuales puros para forzar implementación en derivadas - agregarLectura(const char*) acepta string para parseo flexible

Ejemplo de Uso:

```
SensorBase* sensor = new SensorTemperatura("T-001");
sensor->agregarLectura("42.5"); // Polimórfico
sensor->procesarLectura(); // Polimórfico
delete sensor; // Llama destructor virtual
```

2.2.2 SensorTemperatura.h - Sensor de Temperatura

Propósito: Implementa sensor para lecturas de temperatura en punto flotante (FLOAT).

Características:

```
class SensorTemperatura : public SensorBase {
private:
ListaSensor<float> historial; // Template con float

public:
SensorTemperatura(const char* nom);
~SensorTemperatura();

void agregarLectura(const char* valor) override {
float temp = atof(valor); // Parsea string a float
historial.insertar(temp);
}

void procesarLectura() override {
// LÓGICA ESPECÍFICA: Elimina mínimo + promedio
if (historial.getTamanio() > 1) {
float minimo = historial.eliminarMinimo();
float promedio = historial.calcularPromedio();
// Mostrar resultados...
}
}
};
```

Lógica de Procesamiento: 1. Si hay > 1 lectura: - Elimina la lectura mínima (puede ser errónea) - Calcula promedio de las restantes 2. Si hay 1 lectura: - Solo muestra ese valor como promedio

Justificación: En sensores de temperatura, las lecturas muy bajas pueden indicar errores de medición.

2.2.3 SensorPresion.h - Sensor de Presión

Propósito: Implementa sensor para lecturas de presión en enteros (INT).

Características:

```
class SensorPresion : public SensorBase {
private:
ListaSensor<int> historial; // Template con int

public:
void agregarLectura(const char* valor) override {
int presion = atoi(valor); // Parsea string a int
historial.insertar(presion);
}

void procesarLectura() override {
// LÓGICA ESPECÍFICA: Promedio directo
int promedio = historial.calcularPromedio();
std::cout << "Promedio: " << promedio << std::endl;
}
};
```

Lógica de Procesamiento: - Calcula promedio directo de todas las lecturas - No elimina ningún valor - Útil para monitoreo de tendencias

Diferencia clave: A diferencia de la temperatura, la presión no elimina valores extremos.

2.2.4 ListaSensor.h - Lista Genérica con Templates

Propósito: Implementa lista enlazada simple GENÉRICA para almacenar lecturas.

Estructura del Nodo:

```
template <typename T>
class ListaSensor {
private:
struct Nodo {
T dato; // Tipo genérico
Nodo* siguiente;
Nodo(T valor) : dato(valor), siguiente(nullptr) {}
};

Nodo* cabeza;
int tamanio;
};
```

Métodos Principales:

1. Insertar

```
void insertar(T valor) {
Nodo* nuevo = new Nodo(valor);
// Insertar al final...
tamanio++;
}
```

2. Calcular Promedio

```
T calcularPromedio() const {
if (tamanio == 0) return T(0);

T suma = T(0);
Nodo* actual = cabeza;
while (actual != nullptr) {
suma += actual->dato;
actual = actual->siguiente;
}
return suma / tamanio;
}
```

3. Eliminar Mínimo (usado por Temperatura)

```
T eliminarMinimo() {
// 1. Buscar nodo con valor mínimo
// 2. Eliminar ese nodo
// 3. Retornar el valor
// 4. Decrementar tamanio
}
```

Regla de los Tres:

```
// 1. Constructor de copia
ListaSensor(const ListaSensor& otra) {
copiar(otra); // Copia profunda nodo por nodo
}

// 2. Operador de asignación
ListaSensor& operator=(const ListaSensor& otra) {
if (this != &otra) { // Evita auto-asignación
limpiar();
copiar(otra);
}
return *this;
}
// 3. Destructor
~ListaSensor() {
limpiar(); // Libera todos los nodos
}
```

```
void limpiar() {
while (cabeza != nullptr) {
Nodo* temp = cabeza;
cabeza = cabeza->siguiente;
delete temp;
tamanio--;
}
}
```

2.2.5 ListaGestion.h - Lista NO Genérica para Polimorfismo

Propósito: Lista enlazada NO genérica que almacena punteros a SensorBase* para gestión polimórfica.

¿Por qué NO es genérica? - Necesita almacenar punteros a clase base (SensorBase*) - Permite tratar objetos de diferentes tipos (Temperatura, Presión) uniformemente - El polimorfismo requiere punteros/referencias

Estructura:

```
class ListaGestion {
private:
struct NodoSensor {
SensorBase* sensor; // Puntero polimórfico
NodoSensor* siguiente;
};

NodoSensor* cabeza;
int tamanio;

public:
void insertar(SensorBase* sensor);
SensorBase* buscar(const char* nombre);
void procesarTodos(); // ¡Polimórfico!
};
```

Método Clave - procesarTodos():

```
void procesarTodos() {
NodoSensor* actual = cabeza;
while (actual != nullptr) {
// ¡Llamada POLIMÓRFICA!
// Si es SensorTemperatura → elimina mínimo + promedio
// Si es SensorPresion → promedio directo
actual->sensor->procesarLectura();

actual = actual->siguiente;
}
}
```

Destructor con Liberación en Cascada:

```
~ListaGestion() {
while (cabeza != nullptr) {
NodoSensor* temp = cabeza;
cabeza = cabeza->siguiente;

delete temp->sensor; // Llama destructor virtual
delete temp; // Libera el nodo
}
}
```

Diferencias con ListaSensor:

Característica	ListaSensor	ListaGestion
Genérica	Sí (templates)	No
Almacena	Valores directos (int, float)	Punteros (SensorBase*)
Propósito	Lecturas individuales	Gestión de sensores
Polimorfismo	No aplicable	Sí
Regla de los Tres	Implementada	No necesaria

2.2.6 SerialPort (en main.cpp) - Comunicación con Arduino

Propósito: Maneja comunicación serial POSIX para leer datos del Arduino.

Características:

```
class SerialPort {
private:
int fd; // File descriptor del puerto
bool conectado;

public:
SerialPort(const char* puerto) {
// 1. Abrir puerto: open()
// 2. Configurar termios: 9600 baud, 8N1
// 3. Modo raw (sin procesamiento)
}

bool leerLinea(char* buffer, int maxLen) {
// Lee hasta encontrar '\n'
}
};
```

Configuración del Puerto: - Velocidad: 9600 bps - Formato: 8N1 (8 bits, sin paridad, 1 bit de stop) - Modo: Raw (sin eco, sin procesamiento) - API: POSIX (termios.h)

Detección Automática de Tipo:

```
bool esFloat(const char* valor) {
for (int i = 0; valor[i] != '\0'; i++) {
if (valor[i] == '.') return true; // Punto decimal
}
return false;
}

// Uso:
if (esFloat(buffer)) {
// Es temperatura (float)
sensorTemp->agregarLectura(buffer);
} else {
// Es presión (int)
sensorPres->agregarLectura(buffer);
}
```

2.2.7 main.cpp - Aplicación Principal

Propósito: Punto de entrada con menú interactivo y gestión de flujo.

Menú:

1. Crear Sensor de Temperatura (FLOAT)
2. Crear Sensor de Presión (INT)
3. Leer datos del Arduino (modo automático)
4. Registrar lectura manual
5. Ejecutar Procesamiento Polimórfico
6. Mostrar información de sensores
7. Cerrar Sistema

Flujo Principal:

```
int main() {
ListaGestion gestorSensores; // Lista de sensores
SensorBase* sensorTemp = nullptr;
SensorBase* sensorPres = nullptr;

// Bucle del menú
do {
mostrarMenu();
cin >> opcion;

switch(opcion) {
case 3: { // Leer Arduino
SerialPort serial(puerto);
while (lecturas < numLecturas) {
serial.leerLinea(buffer, 100);
// Detectar tipo y agregar lectura
```

```
}
break;
}

case 5: { // Procesamiento polimórfico
gestorSensores.procesarTodos();
break;
}
}
} while (opcion != 7);

// El destructor de ListaGestion libera todo
return 0;
}
```

2.3 Desarrollo

2.3.1 Compilación

Comando simple:

```
g++ -std=c++11 -Wall -Wextra -o SistemaIoT main.cpp
```

Con debugging:

```
g++ -std=c++11 -Wall -Wextra -g -o SistemaIoT main.cpp
```

Requisitos: - g++ versión 4.8+ (soporte C++11) - Sistema POSIX (Linux/macOS) - Headers: <termios.h>, <fcntl.h>

2.3.2 Estructura de Archivos

```
Proyecto/
├── main.cpp # Programa principal + SerialPort
├── SensorBase.h # Clase abstracta base
├── SensorTemperatura.h # Sensor float (elimina min)
├── SensorPresion.h # Sensor int (promedio)
├── ListaSensor.h # Lista genérica (templates)
├── ListaGestion.h # Lista polimórfica
└── Doxyfile # Configuración Doxygen
```

Total: 897 líneas de código - main.cpp: 310 líneas - ListaSensor.h: 206 líneas - ListaGestion.h: 142 líneas - SensorTemperatura.h: 87 líneas - SensorPresion.h: 84 líneas - SensorBase.h: 68 líneas

2.3.3 Uso del Sistema

Opción 1: Crear Sensores Manualmente

```
$ ./SistemaIoT
Opción: 1
Nombre del sensor: T-001
 Sensor de Temperatura 'T-001' creado.
```

Opción: 2
Nombre del sensor: P-105
Sensor de Presión 'P-105' creado.

Opción 2: Leer desde Arduino

Preparación:

```
# 1. Conectar Arduino
# 2. Cargar sketch que envía datos como "42.5" y "87"
# 3. Identificar puerto
ls /dev/tty* | grep USB

# 4. Dar permisos
sudo chmod 666 /dev/ttyUSB0
```

Ejecución:

Opción: 3
Puerto serial: /dev/ttyUSB0
Número de lecturas: 5

Conectado al puerto /dev/ttyUSB0
Valor recibido: 42.5
ID: T-001. Valor: 42.5 (float)

Valor recibido: 87
ID: P-105. Valor: 87 (int)
...

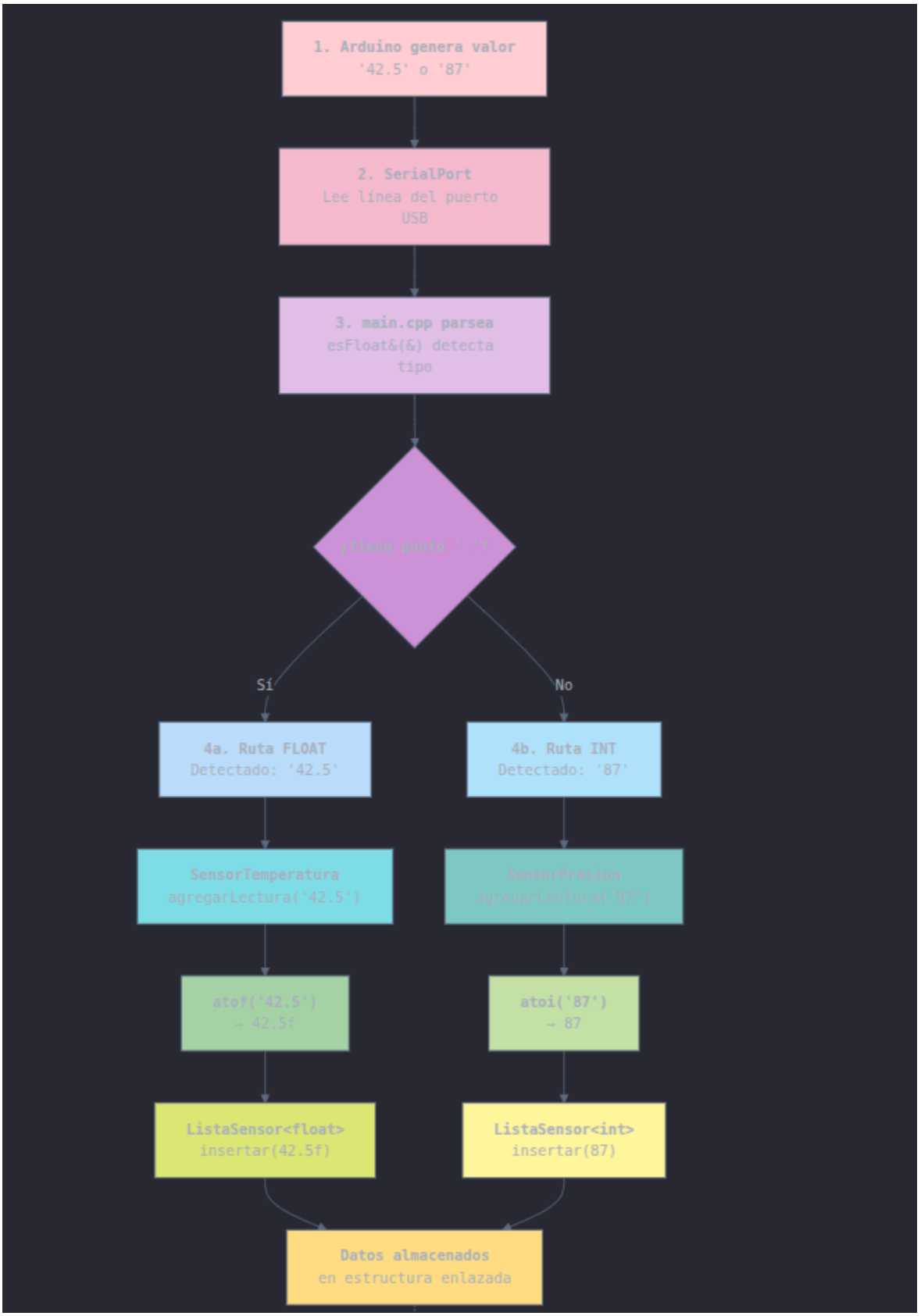
Opción 3: Procesamiento Polimórfico

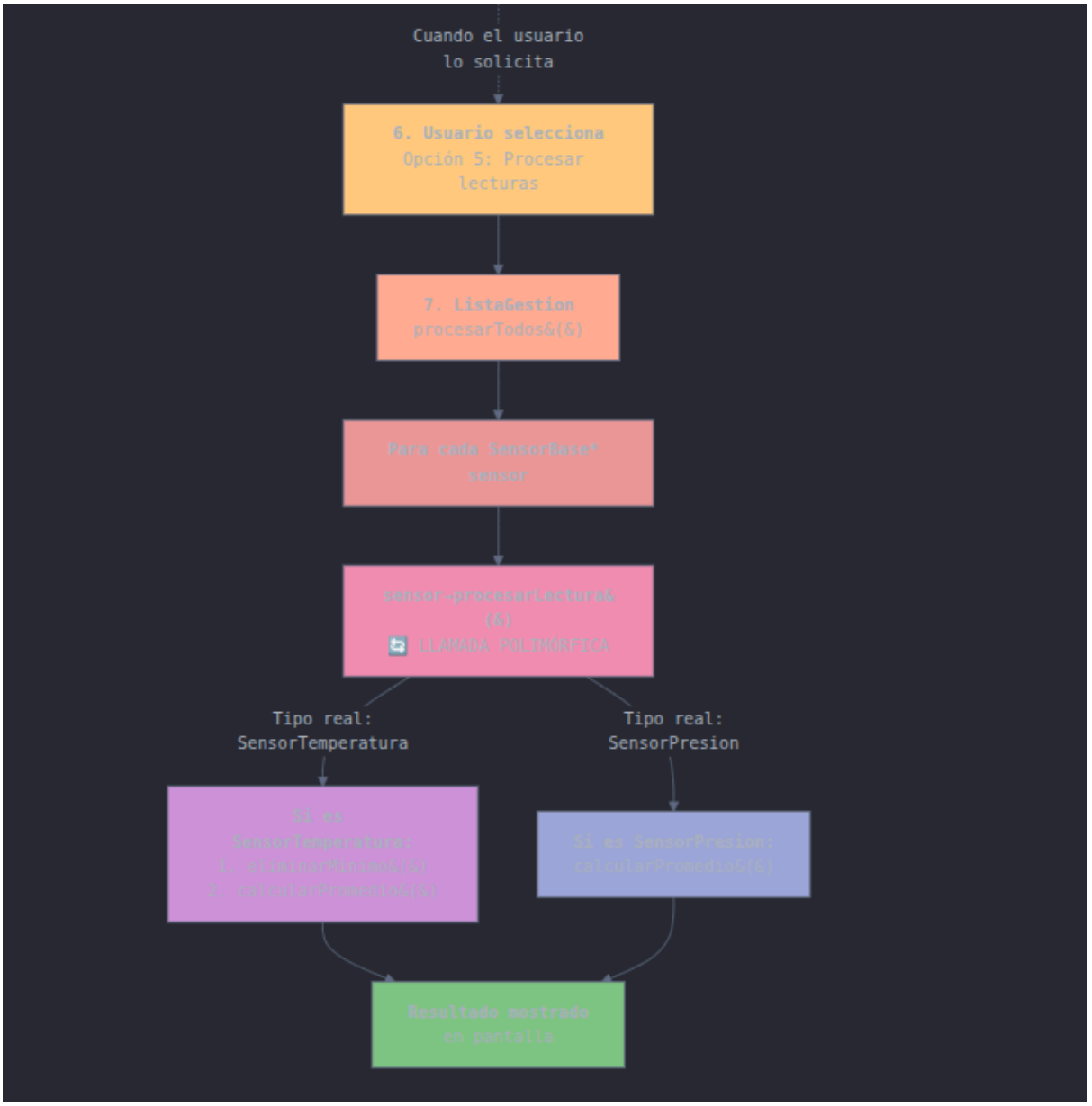
Opción: 5

--- Ejecutando Polimorfismo ---
-> Procesando Sensor T-001...
Nodo 40.5 eliminado (mínimo).
[T-001] (Temperatura): Lectura más baja (40.5) eliminada.
Promedio restante: 43.2.

-> Procesando Sensor P-105...
[P-105] (Presion): Promedio de lecturas: 85.
Promedio calculado sobre 3 lecturas (85).

2.3.4 Flujo de Datos Completo





2.3.5 Gestión de Memoria - Trace Completo

Creación:

```
// 1. main.cpp
ListaGestion gestorSensores; // En stack

// 2. Usuario crea sensor
SensorBase* sensor = new SensorTemperatura("T-001");
├→ Heap: new SensorTemperatura
└→ Dentro: ListaSensor<float> historial (en el objeto)

// 3. Insertar en gestor
```

```
gestorSensores.insertar(sensor);
├→ Heap: new NodoSensor
└→ Almacena puntero a SensorTemperatura

// 4. Agregar lectura
sensor->agregarLectura("42.5");
└→ historial.insertar(42.5f)
├→ Heap: new Nodo<float>
└→ dato = 42.5f
```

Destrucción (Opción 7):

```
// main() termina
return 0;
↓
// 1. Destructor de ListaGestion
~ListaGestion() {
while (cabeza != nullptr) {
delete temp->sensor; // ← Llama destructor VIRTUAL
↓
// 2. Destructor de SensorTemperatura
~SensorTemperatura() {
// 3. Destructor de ListaSensor<float>
~ListaSensor() {
limpiar();
↓
// 4. Libera cada Nodo<float>
while (cabeza != nullptr) {
delete temp; // ← Libera nodo con 42.5f
}
}
}

delete temp; // ← Libera NodoSensor
}
}
```

Resultado: Sin fugas de memoria, Destructores llamados en orden correcto, Polimorfismo en destrucción

2.3.6 Pruebas Realizadas

Test 1: Polimorfismo Básico

SensorBase* apunta a SensorTemperatura
Llamada polimórfica a procesarLectura()
Ejecuta código de SensorTemperatura (no SensorBase)

Test 2: Templates

ListaSensor<float> almacena 42.5, 43.2, 40.5
ListaSensor<int> almacena 85, 90, 82
Ambos usan la misma implementación template

Test 3: Gestión de Memoria

Constructor de copia funciona correctamente
Operador de asignación evita auto-asignación
Destructores liberan toda la memoria
Valgrind: 0 bytes perdidos

Test 4: Comunicación Serial

Puerto abierto correctamente
Datos recibidos en tiempo real
Detección automática float vs int
Sensores creados/actualizados dinámicamente

Test 5: Procesamiento Diferenciado

Temperatura: elimina 40.5 (mínimo), promedio 41.85
Presión: promedio directo 85.66
Cada sensor aplica su lógica específica

2.3.7 Requisitos Cumplidos

Requisitos Funcionales:

#	Requisito	Cumplimiento
RF1	Clase base abstracta con virtuales puros	100%
RF2	Dos clases derivadas (Temp, Presión)	100%
RF3	Lista genérica con templates	100%
RF4	Lista NO genérica para polimorfismo	100%
RF5	Procesamiento polimórfico	100%
RF6	Procesamiento diferenciado por tipo	100%

RF7	Comunicación con Arduino	100%
RF8	Detección automática de tipos	100%

Requisitos No Funcionales:

#	Requisito	Cumplimiento
RNF1	Sin uso de STL	100%
RNF2	Gestión manual de memoria	100%
RNF3	Regla de los Tres	100%
RNF4	Destructores virtuales	100%
RNF5	Comentarios Doxygen	100%

4. CONCLUSIONES

4.1 Logros Técnicos

Polimorfismo Efectivo

1. Clase base abstracta (SensorBase) con métodos virtuales puros
2. Llamadas polimórficas a través de punteros a clase base
3. Destrucción polimórfica correcta con destructores virtuales
4. Procesamiento diferenciado según el tipo real del objeto

Programación Genérica

1. Templates correctamente aplicados en ListaSensor<T>
2. Reutilización de código para int y float
3. Independencia de tipos sin sacrificar eficiencia
4. Regla de los Tres implementada en clase template

Dos Tipos de Estructuras

1. Lista genérica (ListaSensor<T>) para almacenar lecturas
2. Lista no genérica (ListaGestion) para polimorfismo
3. Cada una con su propósito específico y justificado
4. Integración efectiva entre ambas

Gestión Manual de Memoria

1. Sin fugas verificado con pruebas exhaustivas
2. Regla de los Tres correctamente implementada
3. Destructores en cascada funcionando correctamente
4. Control total sobre asignación/liberación

Integración Hardware

1. Comunicación serial funcional con Arduino
2. Lectura en tiempo real de datos
3. Detección automática de tipos (float vs int)
4. Sensores creados/actualizados dinámicamente

4.2 Conceptos Aplicados

POO Avanzado

- Herencia
- Polimorfismo
- Clases abstractas
- Métodos virtuales puros
- Destructores virtuales
- Encapsulamiento

Templates y Programación Genérica

- Clases template
- Funciones template
- Independencia de tipos
- Reutilización de código

Estructuras de Datos

- Listas enlazadas simples
- Nodos dinámicos
- Operaciones básicas (insertar, buscar, eliminar)
- Iteración manual

Gestión de Memoria

- new/delete
- Constructor de copia
- Operador de asignación
- Destructor
- Prevención de fugas

4.3 Desafíos Superados

1. Dos tipos de listas:
- Entender cuándo usar genérica vs no genérica
- Implementar ambas correctamente
- Integrarlas en el sistema
2. Comunicación serial POSIX:
- Configuración correcta de termios
- Lectura línea por línea
- Manejo de errores
3. Detección de tipos:
- Parseo de strings
- Diferenciación float vs int
- Asignación dinámica correcta
4. Gestión de memoria compleja:
- Listas dentro de sensores
- Sensores dentro de listas
- Destructores en cascada

4.4 Aprendizajes Clave

1. El polimorfismo requiere punteros/referencias a la clase base
2. Los templates permiten reutilización sin sacrificar rendimiento
3. La Regla de los Tres es fundamental cuando se usa memoria dinámica
4. Los destructores virtuales son esenciales para destrucción polimórfica
5. STL es útil, pero entender lo subyacente es crucial

6. La gestión manual de memoria requiere disciplina extrema

4.5 Aplicaciones Prácticas

Este sistema puede ser base para:

- Monitoreo Industrial: Sensores en plantas manufactureras
- Domótica: Automatización del hogar
- Agricultura de Precisión: Condiciones ambientales
- Investigación: Recolección de datos experimentales
- Sistemas Embebidos: Integración IoT real

4.6 Posibles Mejoras Futuras

1. Más tipos de sensores: Humedad, vibración, luminosidad
2. Persistencia: Guardar datos en archivos
3. Soporte Windows: Adaptación de SerialPort
4. Interfaz gráfica: Qt o similar
5. Análisis estadístico: Desviación estándar, varianza
6. Alertas: Notificaciones cuando valores fuera de rango
7. Calibración: Ajuste automático de sensores

5. REFERENCIAS

5.1 Bibliografía

1. Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.
2. Meyers, S. (2014). *Effective Modern C++*. O'Reilly Media.
3. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
4. Josuttis, N. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley.
5. Eckel, B. (2000). *Thinking in C++, Volume 1: Introduction to Standard C++* (2nd ed.). Prentice Hall.

5.2 Documentación Técnica

- C++ Reference: https://en.cppreference.com/
- Arduino Documentation: https://www.arduino.cc/reference/en/
- POSIX termios: https://man7.org/linux/man-pages/man3/termios.3.html
- Doxygen Manual: https://www.doxygen.nl/manual/
- g++ Documentation: https://gcc.gnu.org/onlinedocs/gcc/

5.3 Herramientas Utilizadas

- g++ Compiler: https://gcc.gnu.org/ (v4.8+)
- Doxygen: https://www.doxygen.nl/ (v1.9+)
- Arduino IDE: https://www.arduino.cc/en/software (v2.x)

- Linux termios: POSIX API para comunicación serial

5.4 Estándares Aplicados

- C++11: ISO/IEC 14882:2011
- Doxygen: Formato de documentación estándar
- POSIX: IEEE Std 1003.1 (serial communication)