

Sistema IoT de Gestión Polimórfica de Sensores

Manual Técnico

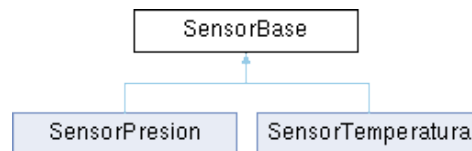
Listas Enlazadas Simples y Polimorfismo

Autor: FabiRamiro

Materia: Estructura de Datos

Actividad: Listas Enlazadas Simples - Unidad 2

31 de Octubre, 2025



Índice

Índice de figuras

1 Introducción

1.1 Contexto del Problema

Las empresas de monitoreo de Infraestructura Crítica (IC) enfrentan el desafío de gestionar múltiples tipos de sensores que generan datos heterogéneos. Estos sistemas requieren:

- **Flexibilidad de tipos de datos:** Los sensores pueden generar lecturas enteras (conteo de eventos) o de punto flotante (mediciones continuas).
- **Escalabilidad:** El número de sensores y lecturas varía dinámicamente.
- **Procesamiento unificado:** Diferentes tipos de sensores deben poder procesarse mediante una interfaz común.
- **Eficiencia de memoria:** Gestión manual sin sobrecarga de bibliotecas estándar.

El presente proyecto desarrolla un sistema de bajo nivel que supera estas limitaciones mediante el uso de **Programación Orientada a Objetos Avanzada y Estructuras de Datos Genéricas**.

1.2 Objetivos del Proyecto

1.2.1. Objetivo General

Diseñar e implementar un sistema de gestión polimórfica de sensores IoT que integre listas enlazadas genéricas, comunicación serial con hardware ESP32 y gestión manual de memoria dinámica.

1.2.2. Objetivos Específicos

1. Implementar una jerarquía de clases polimórfica con clase base abstracta (**SensorBase**).
2. Desarrollar listas enlazadas genéricas (**ListaSensor<T>**) sin usar STL.
3. Integrar comunicación serial con ESP32 mediante Windows API.
4. Aplicar la Regla de los Tres para gestión correcta de memoria.
5. Demostrar polimorfismo en tiempo de ejecución.
6. Documentar el código completamente usando Doxygen.

1.3 Alcance del Sistema

El sistema permite:

- **Registro dinámico** de sensores de temperatura (lecturas `float`) y presión (lecturas `int`).
- **Captura en tiempo real** de datos desde dispositivo ESP32 vía puerto serial (COM).

- **Almacenamiento** de lecturas en listas enlazadas simples genéricas.
- **Procesamiento polimórfico** de sensores con algoritmos específicos:
 - Temperatura: Elimina lectura mínima y calcula promedio.
 - Presión: Calcula promedio de todas las lecturas.
- **Liberación en cascada** de memoria sin fugas.

1.4 Justificación

Este proyecto integra conceptos fundamentales de:

- **POO Avanzada:** Clases abstractas, herencia, polimorfismo, destructores virtuales.
- **Estructuras de Datos:** Listas enlazadas, gestión manual de nodos.
- **Programación Genérica:** Templates en C++.
- **Gestión de Memoria:** Regla de los Tres/Cinco.
- **Sistemas Embebidos:** Comunicación serial con microcontroladores.

La prohibición de usar STL (`std::list`, `std::vector`) obliga a comprender profundamente la gestión de memoria y punteros en C++, habilidades esenciales para sistemas de bajo nivel y tiempo real.

1.5 Tecnologías Utilizadas

Tabla 1: Tecnologías y herramientas del proyecto

Categoría	Tecnología
Lenguaje de programación	C++11
Sistema de compilación	CMake 3.10+
Compilador	MinGW-w64 / GCC
Documentación	Doxygen 1.9+
Microcontrolador	ESP32 Dev Module
Entorno de desarrollo	VS Code, Arduino IDE
Sistema operativo	Windows 10/11
Control de versiones	Git / GitHub Classroom
Comunicación serial	Windows API (CreateFile, ReadFile)

1.6 Estructura del Documento

Este manual técnico se organiza de la siguiente manera:

- **Sección 2 - Diseño:** Arquitectura del sistema, diagramas de clases, patrones de diseño.

- **Sección 3 - Desarrollo:** Implementación de conceptos de POO, gestión de memoria, comunicación serial.
- **Sección 4 - Componentes:** Descripción detallada de cada clase del sistema.
- **Sección 5 - Pantallazos:** Capturas de implementación, compilación y ejecución.
- **Sección 6 - Conclusiones:** Logros, dificultades y aprendizajes.

2 Diseño del Sistema

2.1 Arquitectura General

El sistema se estructura en cuatro capas principales que separan responsabilidades y facilitan el mantenimiento:



Figura 1: Arquitectura del sistema en capas

2.1.1. Capa de Abstracción (POO)

Define la jerarquía polimórfica de sensores:

- **SensorBase:** Clase base abstracta con métodos virtuales puros.
- **SensorTemperatura:** Implementación concreta para sensores de temperatura.
- **SensorPresion:** Implementación concreta para sensores de presión.

2.1.2. Capa de Estructuras de Datos

Gestiona el almacenamiento dinámico de información:

- **ListaSensor<T>:** Plantilla genérica de lista enlazada simple.
- **ListaGeneral:** Lista polimórfica que almacena punteros **SensorBase***.

2.1.3. Capa de Comunicación

Interfaz con hardware externo:

- **SerialReader:** Maneja comunicación serial con ESP32 mediante Windows API.

2.1.4. Capa de Aplicación

Interfaz de usuario y lógica principal:

- **main.cpp:** Menú interactivo y coordinación de componentes.

2.2 Diagrama de Clases

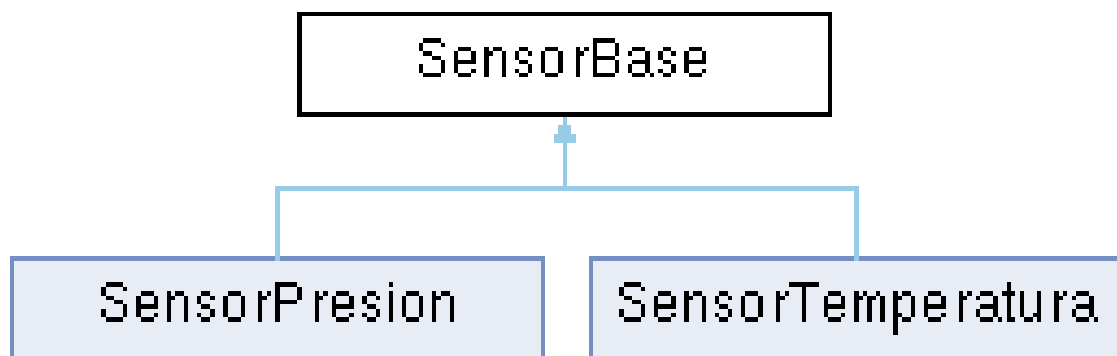


Figura 2: Diagrama de jerarquía de clases (generado por Doxygen)

La figura ?? muestra la relación de herencia entre clases. **SensorBase** define la interfaz común que todas las clases derivadas deben implementar.

2.3 Relaciones entre Componentes

```
1 class SensorTemperatura : public SensorBase {
2 private:
3     ListaSensor<float> historial; // Composicion
4     // ...
5 };
6
7 class SensorPresion : public SensorBase {
8 private:
9     ListaSensor<int> historial; // Composicion
10    // ...
11};
```

Listing 1: Relación de composición

Cada sensor **contiene** (composición) una lista enlazada del tipo apropiado para sus lecturas.

2.4 Patrones de Diseño Implementados

2.4.1. Template Method Pattern

El método `procesarLectura()` en `SensorBase` define un algoritmo esqueleto, permitiendo que las subclases implementen pasos específicos:

```
1 // SensorBase.h (clase abstracta)
2 virtual void procesarLectura() = 0; // Template Method
3
4 // SensorTemperatura.cpp
5 void SensorTemperatura::procesarLectura() {
6     // Implementacion especifica para temperatura
7     float minimo = historial.eliminarMinimo();
8     float promedio = historial.calcularPromedio();
9     // ...
10 }
11
12 // SensorPresion.cpp
13 void SensorPresion::procesarLectura() {
14     // Implementacion especifica para presion
15     float promedio = historial.calcularPromedio();
16     // ...
17 }
```

Listing 2: Patrón Template Method

2.4.2. Factory Pattern (Simplificado)

La creación dinámica de sensores al recibir datos del ESP32 implementa un patrón Factory simplificado:

```
1 // main.cpp
2 if (std::strcmp(tipo, "TEMP") == 0) {
3     sensor = new SensorTemperatura(id);
4 } else if (std::strcmp(tipo, "PRES") == 0) {
5     sensor = new SensorPresion(id);
6 }
```

```
7 sistemaGestion.insertarSensor(sensor);
```

Listing 3: Creación dinámica de sensores

2.4.3. RAII (Resource Acquisition Is Initialization)

Los destructores automáticos garantizan liberación de recursos:

```
1 {  
2     ListaGeneral sistema; // Constructor adquiere recursos  
3     sistema.insertarSensor(new SensorTemperatura("T-001"));  
4     // ...  
5 } // Destructor libera automaticamente TODA la memoria
```

Listing 4: RAII en acción

2.5 Protocolo de Comunicación Serial

El ESP32 envía datos en formato texto plano:

TIPO:ID:VALOR\n

Ejemplos:

- TEMP:T-001:23.5 → Temperatura de 23.5°C
- PRES:P-105:85 → Presión de 85 PSI

Especificaciones:

- Baud rate: 115200
- Data bits: 8
- Stop bits: 1
- Parity: None
- Terminador de línea: \n

2.6 Gestión de Memoria

2.6.1. Diagrama de Flujo de Liberación

La liberación de memoria ocurre en cascada:

1. El destructor de `ListaGeneral` se ejecuta.
2. Itera sobre cada nodo de la lista principal.
3. Para cada nodo, llama a `delete sensor;` (puntero `SensorBase*`).
4. Por polimorfismo, se ejecuta el destructor de la clase derivada (`SensorTemperatura` o `SensorPresion`).

5. El destructor derivado libera su `ListaSensor<T>` interno.
6. El destructor de `ListaSensor<T>` libera todos sus nodos.

Importancia del destructor virtual:

```
1 class SensorBase {  
2 public:  
3     virtual ~SensorBase(); // CRUCIAL para polimorfismo  
4     // ...  
5 };
```

Listing 5: Destructor virtual en clase base

Sin el destructor virtual, al ejecutar `delete sensorBase*`, solo se llamaría al destructor de `SensorBase`, causando **fugas de memoria** en las listas internas.

3 Desarrollo e Implementación

3.1 Conceptos de POO Aplicados

3.1.1. Polimorfismo

El polimorfismo permite tratar objetos de diferentes clases de manera uniforme a través de una interfaz común.

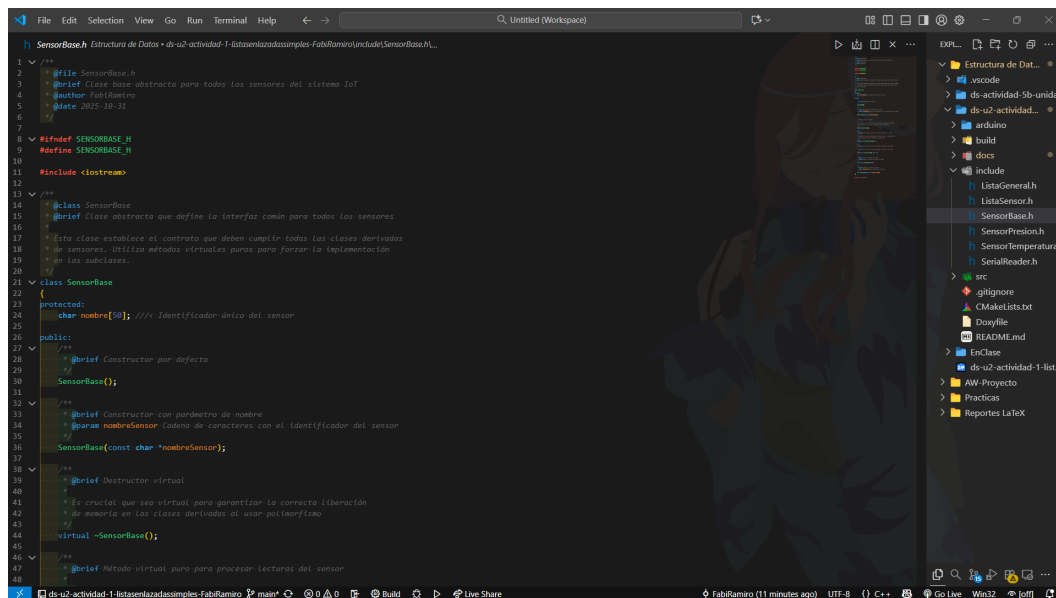


Figura 3: Declaración de clase base abstracta con métodos virtuales puros

Implementación:

```

1 class SensorBase {
2 protected:
3     char nombre[50];
4
5 public:
6     SensorBase();
7     SensorBase(const char* nombreSensor);
8
9     // Destructor virtual (OBLIGATORIO)
10    virtual ~SensorBase();
11
12    // Metodos virtuales puros
13    virtual void procesarLectura() = 0;
14    virtual void imprimirInfo() const = 0;
15
16    const char* getNombre() const;
17    void setNombre(const char* nombreSensor);
18 };

```

Listing 6: Métodos virtuales puros en SensorBase.h

Uso polimórfico:

```

1 void ListaGeneral::procesarTodosSensores() {
2     NodoSensor* actual = cabeza;

```

```

3
4 while (actual != nullptr) {
5     // Llamada polimorfica: el metodo correcto se
6     // determina en tiempo de ejecucion segun el
7     // tipo real del objeto (SensorTemperatura o SensorPresion)
8     actual->sensor->procesarLectura();
9
10    actual = actual->siguiente;
11 }
12 }

```

Listing 7: Procesamiento polimórfico en ListaGeneral.cpp

3.1.2. Templates (Programación Genérica)

Los templates permiten escribir código independiente del tipo de datos.

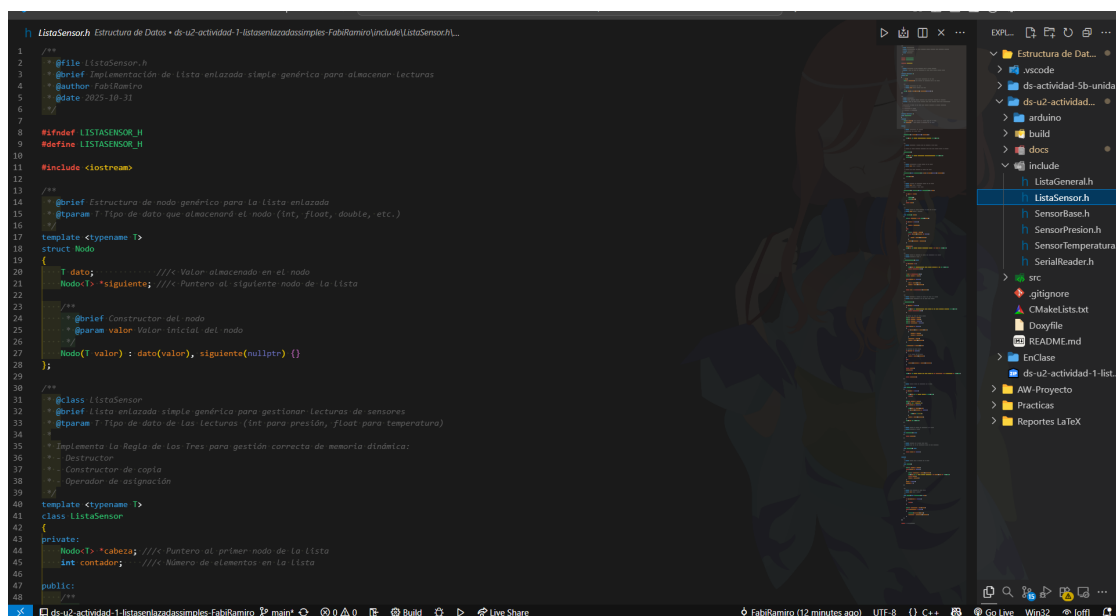


Figura 4: Implementación de lista enlazada genérica con template

Declaración del template:

```

1 template <typename T>
2 struct Nodo {
3     T dato;
4     Nodo<T>* siguiente;
5
6     Nodo(T valor) : dato(valor), siguiente(nullptr) {}
7 };

```

Listing 8: Nodo genérico en ListaSensor.h

```

1 template <typename T>
2 class ListaSensor {
3 private:
4     Nodo<T>* cabeza;
5     int contador;
6 }

```

```

7 public:
8     ListaSensor();
9     ~ListaSensor();
10    ListaSensor(const ListaSensor<T>& otra);
11    ListaSensor<T>& operator=(const ListaSensor<T>& otra);
12
13    void insertar(T valor);
14    T calcularPromedio() const;
15    T eliminarMinimo();
16    int getContador() const;
17    bool estaVacia() const;
18 };

```

Listing 9: Clase ListaSensor<T>

Instanciación del template:

```

1 // En SensorTemperatura.h
2 class SensorTemperatura : public SensorBase {
3 private:
4     ListaSensor<float> historial; // Template instanciado con float
5     // ...
6 };
7
8 // En SensorPresion.h
9 class SensorPresion : public SensorBase {
10 private:
11     ListaSensor<int> historial; // Template instanciado con int
12     // ...
13 };

```

Listing 10: Uso de templates en sensores

El compilador genera dos versiones especializadas del código de `ListaSensor`: una para `float` y otra para `int`.

3.1.3. Herencia

```

1 class SensorTemperatura : public SensorBase {
2 private:
3     ListaSensor<float> historial;
4
5 public:
6     SensorTemperatura(const char* nombreSensor);
7     ~SensorTemperatura();
8
9     void registrarLectura(float temperatura);
10
11     // Implementacion de metodos virtuales puros
12     void procesarLectura() override;
13     void imprimirInfo() const override;
14 };

```

Listing 11: Herencia en SensorTemperatura.h

La palabra clave `override` (C++11) verifica que efectivamente se está sobrescribiendo un método virtual de la clase base.

3.2 Gestión de Memoria: Regla de los Tres

La Regla de los Tres establece que si una clase necesita un destructor personalizado, probablemente también necesita:

1. Destructor
2. Constructor de copia
3. Operador de asignación

3.2.1. Destructor

```
1 template <typename T>
2 ListaSensor<T>::~~ListaSensor() {
3     std::cout << "[Log] Destruyendo ListaSensor<T>..." << std::endl;
4     limpiar();
5 }
6
7 template <typename T>
8 void ListaSensor<T>::limpiar() {
9     Nodo<T>* actual = cabeza;
10    while (actual != nullptr) {
11        Nodo<T>* siguiente = actual->siguiente;
12        std::cout << " [Log] Nodo<T> liberado: "
13                << actual->dato << std::endl;
14        delete actual;
15        actual = siguiente;
16    }
17    cabeza = nullptr;
18    contador = 0;
19 }
```

Listing 12: Destructor de ListaSensor<T>

3.2.2. Constructor de Copia

```
1 template <typename T>
2 ListaSensor<T>::ListaSensor(const ListaSensor<T>& otra)
3     : cabeza(nullptr), contador(0) {
4     copiar(otra);
5 }
6
7 template <typename T>
8 void ListaSensor<T>::copiar(const ListaSensor<T>& otra) {
9     if (otra.cabeza == nullptr) {
10        return;
11    }
12
13    Nodo<T>* actualOtra = otra.cabeza;
14    while (actualOtra != nullptr) {
15        insertar(actualOtra->dato); // Copia profunda
16        actualOtra = actualOtra->siguiente;
17    }
18 }
```

Listing 13: Constructor de copia de ListaSensor<T>

3.2.3. Operador de Asignación

```
1 template <typename T>
2 ListaSensor<T>& ListaSensor<T>::operator=(const ListaSensor<T>& otra)
3 {
4     if (this != &otra) { // Proteccion contra autoasignacion
5         limpiar();
6         copiar(otra);
7     }
8     return *this;
9 }
```

Listing 14: Operador de asignación de ListaSensor<T>

3.3 Comunicación Serial con ESP32

3.3.1. Implementación en Windows

La clase SerialReader encapsula la comunicación serial usando Windows API:

```
1 class SerialReader {
2 private:
3     bool conectado;
4     HANDLE hSerial; // Handle de Windows
5     COMSTAT status;
6     DWORD errors;
7     char puertoActual[20];
8
9 public:
10    SerialReader();
11    ~SerialReader();
12
13    bool conectar(const char* puerto, int baudRate = 115200);
14    void desconectar();
15    bool leerLinea(char* buffer, int tamano, int timeout = 5000);
16    int bytesDisponibles();
17    bool estaConectado() const;
18 };
```

Listing 15: Declaración de SerialReader.h

3.3.2. Apertura del Puerto Serial

```
1 bool SerialReader::conectar(const char* puerto, int baudRate) {
2     // Formato especial para puertos COM > 9
3     char nombrePuerto[20];
4     snprintf(nombrePuerto, sizeof(nombrePuerto), "\\\\.\\%s", puerto);
5
6     // Abrir puerto
7     hSerial = CreateFileA(nombrePuerto,
8                           GENERIC_READ | GENERIC_WRITE,
9                           0, NULL, OPEN_EXISTING,
10                          FILE_ATTRIBUTE_NORMAL, NULL);
11
12     if (hSerial == INVALID_HANDLE_VALUE) {
13         return false;
14     }
15 }
```



```
14     }
15
16     // Configurar parametros (DCB)
17     DCB dcbSerialParams = {0};
18     dcbSerialParams.DCBlength = sizeof(dcbSerialParams);
19     GetCommState(hSerial, &dcbSerialParams);
20
21     dcbSerialParams.BaudRate = baudRate;
22     dcbSerialParams.ByteSize = 8;
23     dcbSerialParams.StopBits = ONESTOPBIT;
24     dcbSerialParams.Parity = NOPARITY;
25
26     SetCommState(hSerial, &dcbSerialParams);
27
28     conectado = true;
29     return true;
30 }
```

Listing 16: Conexión al puerto COM (SerialReader.cpp)

3.3.3. Lectura de Datos

```
1 bool SerialReader::leerLinea(char* buffer, int tamano, int timeout) {
2     int index = 0;
3     char c;
4     DWORD bytesRead;
5     DWORD startTime = GetTickCount();
6
7     while (index < tamano - 1) {
8         if ((GetTickCount() - startTime) > (DWORD)timeout) {
9             return false; // Timeout
10        }
11
12        if (!ReadFile(hSerial, &c, 1, &bytesRead, NULL)) {
13            return false;
14        }
15
16        if (bytesRead == 0) {
17            Sleep(10);
18            continue;
19        }
20
21        if (c == '\n') break; // Fin de linea
22        if (c != '\r') {
23            buffer[index++] = c; // Agregar caracter
24        }
25    }
26
27    buffer[index] = '\0';
28    return index > 0;
29 }
```

Listing 17: Lectura de línea completa desde serial

3.4 Compilación con CMake

El archivo `CMakeLists.txt` define el proceso de compilación:

```
1 cmake_minimum_required(VERSION 3.10)
2 project(SistemaIoTSensores VERSION 1.0 LANGUAGES CXX)
3
4 set(CMAKE_CXX_STANDARD 11)
5 set(CMAKE_CXX_STANDARD_REQUIRED True)
6
7 include_directories(${PROJECT_SOURCE_DIR}/include)
8
9 set(SOURCES
10     src/main.cpp
11     src/SensorBase.cpp
12     src/SensorTemperatura.cpp
13     src/SensorPresion.cpp
14     src/ListaGeneral.cpp
15     src/SerialReader.cpp
16 )
17
18 add_executable(${PROJECT_NAME} ${SOURCES})
```

Listing 18: `CMakeLists.txt`

Proceso de compilación:

```
mkdir build
cd build
cmake .. -G "MinGW Makefiles"
mingw32-make
```

4 Componentes del Sistema

4.1 SensorBase (Clase Abstracta)

Archivos: include/SensorBase.h, src/SensorBase.cpp

4.1.1. Propósito

Define la interfaz común para todos los tipos de sensores mediante métodos virtuales puros, garantizando que todas las subclases implementen el comportamiento mínimo requerido.

4.1.2. Atributos

```
1 protected:
2     char nombre[50]; // Identificador unico del sensor
```

Se usa char[] en lugar de std::string para cumplir con la restricción de no usar STL.

4.1.3. Métodos Públicos

Tabla 2: Métodos de SensorBase

Método	Descripción
SensorBase()	Constructor por defecto
SensorBase(const char*)	Constructor con nombre
virtual ~SensorBase()	Destructor virtual
virtual void procesarLectura() = 0	Método virtual puro de procesamiento
virtual void imprimirInfo() const = 0	Método virtual puro de impresión
const char* getNombre() const	Obtiene el nombre del sensor
void setNombre(const char*)	Establece el nombre del sensor

4.1.4. Diagrama UML

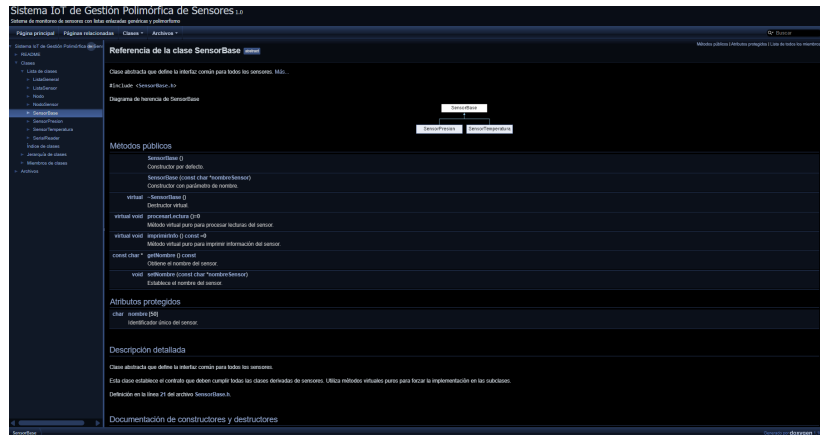


Figura 5: Diagrama UML de SensorBase (Doxygen)

4.2 SensorTemperatura

Archivos: include/SensorTemperatura.h, src/SensorTemperatura.cpp

4.2.1. Propósito

Implementación concreta de sensor que maneja lecturas de temperatura en grados Celsius como valores de punto flotante.

4.2.2. Atributos Privados

```
1 private:
2     ListaSensor<float> historial; // Lista de lecturas
```

4.2.3. Métodos Públicos

```
1 public:
2     SensorTemperatura(const char* nombreSensor);
3     ~SensorTemperatura();
4
5     void registrarLectura(float temperatura);
6     void procesarLectura() override;
7     void imprimirInfo() const override;
```

4.2.4. Algoritmo de Procesamiento

1. Verificar que existan lecturas
2. Eliminar el valor mínimo del historial
3. Calcular el promedio de los valores restantes
4. Imprimir resultado

```

1 void SensorTemperatura::procesarLectura() {
2     if (historial.estaVacia()) {
3         std::cout << "No hay lecturas" << std::endl;
4         return;
5     }
6
7     float minimo = historial.eliminarMinimo();
8     std::cout << "Lectura minima (" << minimo
9         << " C) eliminada." << std::endl;
10
11     if (!historial.estaVacia()) {
12         float promedio = historial.calcularPromedio();
13         std::cout << "Promedio: " << promedio
14             << " C" << std::endl;
15     }
16 }

```

Listing 19: Implementación de procesarLectura()

4.3 SensorPresion

Archivos: include/SensorPresion.h, src/SensorPresion.cpp

4.3.1. Propósito

Implementación concreta de sensor que maneja lecturas de presión en PSI como valores enteros.

4.3.2. Atributos Privados

```

1 private:
2     ListaSensor<int> historial; // Lista de lecturas

```

4.3.3. Diferencias con SensorTemperatura

Tabla 3: Comparación entre SensorTemperatura y SensorPresion

Aspecto	SensorTemperatura	SensorPresion
Tipo de dato	float	int
Unidad	Grados Celsius (°C)	PSI
Procesamiento	Elimina mínimo + promedio	Solo promedio
Template usado	ListaSensor<float>	ListaSensor<int>

4.4 ListaSensor<T>(Template Genérico)

Archivo: include/ListaSensor.h

4.4.1. Propósito

Lista enlazada simple genérica que puede almacenar lecturas de cualquier tipo numérico (int, float, double, etc.).

4.4.2. Estructura del Nodo

```

1 template <typename T>
2 struct Nodo {
3     T dato;
4     Nodo<T>* siguiente;
5
6     Nodo(T valor) : dato(valor), siguiente(nullptr) {}
7 };

```

4.4.3. Atributos Privados

```

1 private:
2     Nodo<T>* cabeza;    // Puntero al primer nodo
3     int contador;       // Numero de elementos

```

4.4.4. Operaciones Principales

Tabla 4: Operaciones de ListaSensor<T>

Operación	Complejidad	Descripción
insertar(T valor)	$O(n)$	Inserta al final
calcularPromedio()	$O(n)$	Retorna promedio
eliminarMinimo()	$O(n)$	Encuentra y elimina mínimo
estaVacía()	$O(1)$	Verifica si está vacía
getContador()	$O(1)$	Retorna número de elementos



Figura 6: Documentación de ListaSensor<T> en Doxygen

4.4.5. Implementación de la Regla de los Tres

```

1 // 1. Destructor
2 ~ListaSensor() {
3     limpiar();
4 }
5
6 // 2. Constructor de copia
7 ListaSensor(const ListaSensor<T>& otra)
8     : cabeza(nullptr), contador(0) {
9     copiar(otra);
10 }
11
12 // 3. Operador de asignacion
13 ListaSensor<T>& operator=(const ListaSensor<T>& otra) {
14     if (this != &otra) {
15         limpiar();
16         copiar(otra);
17     }
18     return *this;
19 }

```

Listing 20: Regla de los Tres en ListaSensor<T>

4.5 ListaGeneral

Archivos: include/ListaGeneral.h, src/ListaGeneral.cpp

4.5.1. Propósito

Lista enlazada simple NO genérica que almacena punteros polimórficos a **SensorBase**, permitiendo gestionar diferentes tipos de sensores en una única estructura.

4.5.2. Estructura del Nodo

```

1 struct NodoSensor {
2     SensorBase* sensor;      // Puntero polimorfico
3     NodoSensor* siguiente;
4
5     NodoSensor(SensorBase* s) : sensor(s), siguiente(nullptr) {}
6 };

```

4.5.3. Operaciones Principales

```

1 public:
2     ListaGeneral();
3     ~ListaGeneral();
4
5     void insertarSensor(SensorBase* sensor);
6     SensorBase* buscarSensor(const char* nombre) const;
7     void procesarTodosSensores();
8     void imprimirTodosSensores() const;
9     int getContador() const;

```

4.5.4. Ejemplo de Polimorfismo

```

1 void ListaGeneral::procesarTodosSensores() {
2     NodoSensor* actual = cabeza;
3
4     while (actual != nullptr) {
5         // El metodo correcto se determina en tiempo de ejecucion
6         // segun el tipo real del objeto apuntado
7         actual->sensor->procesarLectura();
8
9         actual = actual->siguiente;
10    }
11 }

```

Listing 21: Procesamiento polimórfico

En tiempo de ejecución, si `actual->sensor` apunta a:

- `SensorTemperatura*` → Se ejecuta `SensorTemperatura::procesarLectura()`
- `SensorPresion*` → Se ejecuta `SensorPresion::procesarLectura()`

4.6 SerialReader

Archivos: `include/SerialReader.h`, `src/SerialReader.cpp`

4.6.1. Propósito

Encapsula la comunicación serial con el ESP32 usando la API nativa de Windows (`windows.h`).

4.6.2. Atributos Privados (Windows)

```

1 private:
2     bool conectado;
3     HANDLE hSerial;           // Handle del puerto COM
4     COMSTAT status;          // Estado de comunicacion
5     DWORD errors;             // Errores
6     char puertoActual[20];    // Nombre del puerto (ej: "COM3")

```

4.6.3. Configuración del Puerto

Tabla 5: Parámetros de comunicación serial

Parámetro	Valor
Baud rate	115200
Data bits	8
Stop bits	1
Parity	None
Flow control	None
Timeout lectura	5000 ms

4.7 main.cpp (Programa Principal)

Archivo: src/main.cpp

4.7.1. Funciones Auxiliares

```

1 void imprimirMenu();
2 void parsearLinea(const char* linea, char* tipo,
3                  char* id, char* valor);
4 void detectarPuertosCOM();

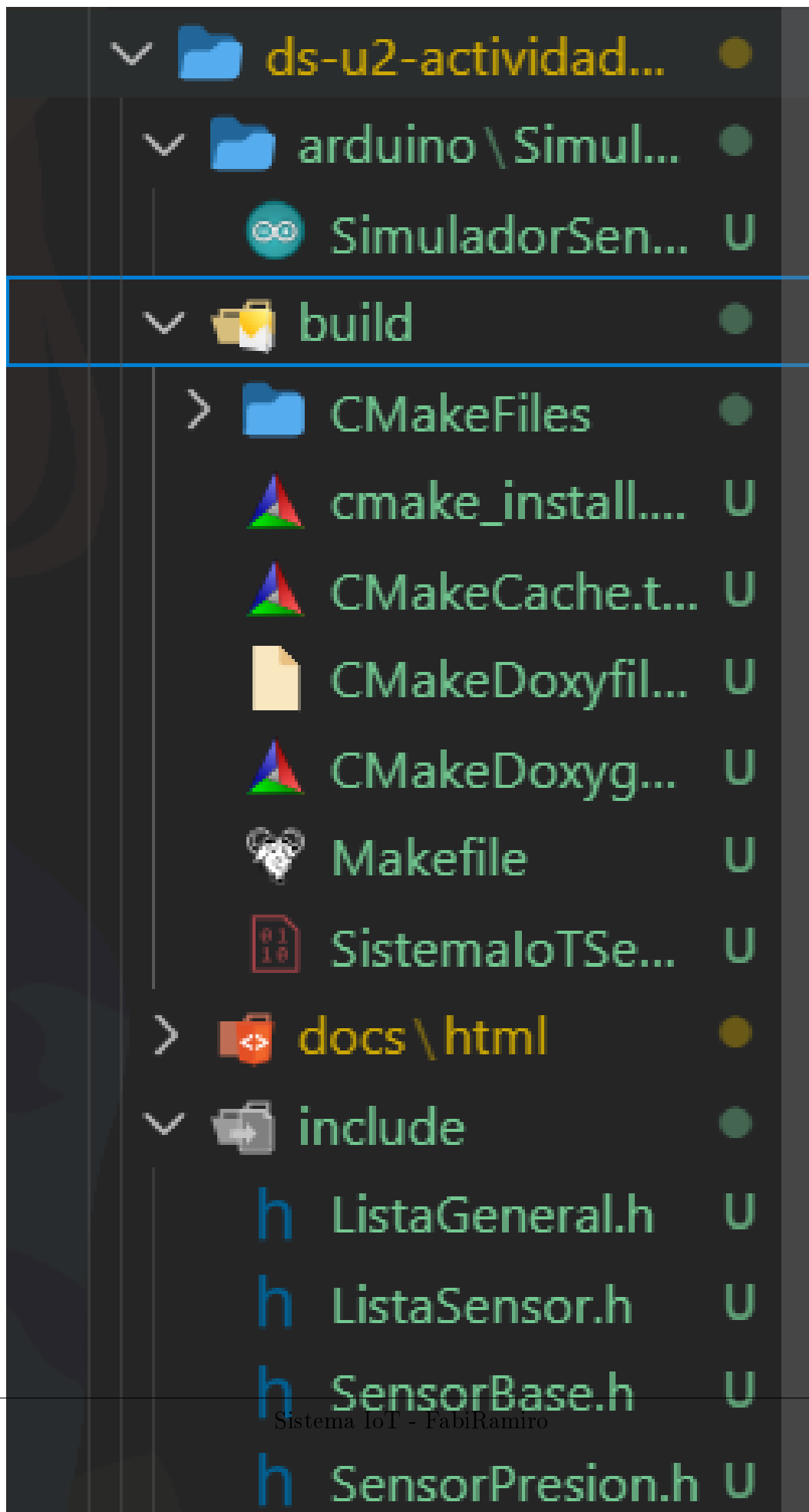
```

4.7.2. Flujo Principal

1. Crear `ListaGeneral` y `SerialReader`
2. Mostrar menú de opciones
3. Procesar selección del usuario:
 - Crear sensores manualmente
 - Conectar a ESP32 y capturar datos
 - Procesar sensores polimórficamente
 - Mostrar información
4. Al salir, el destructor de `ListaGeneral` libera toda la memoria

5 Capturas de Implementación

5.1 Compilación del Proyecto



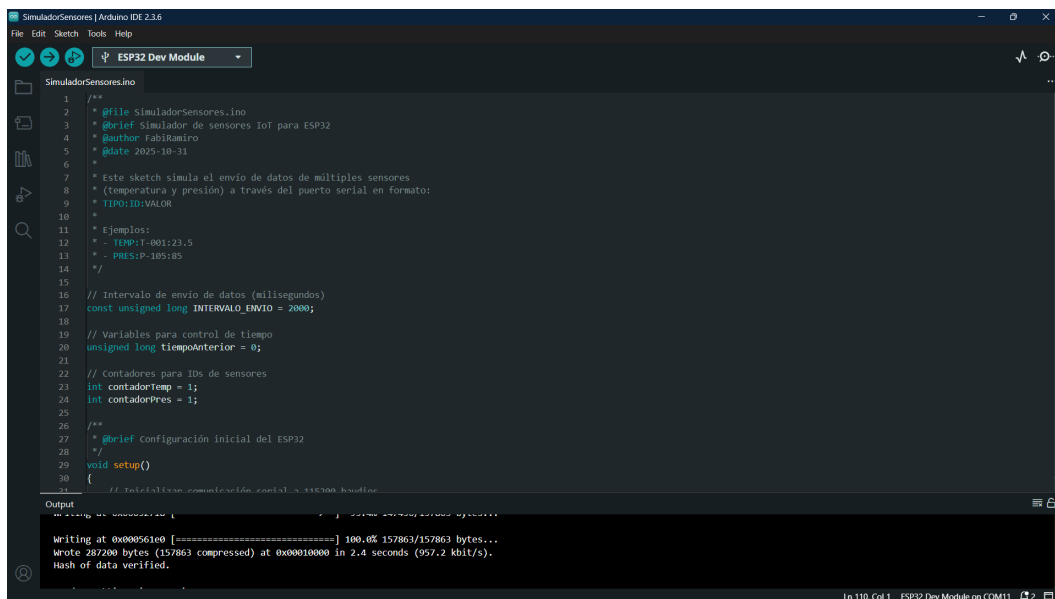
```
fabi7@fabian MINGW64 ~/OneDrive/Documentos/Estructura de Datos/ds-u2-actividad-1-listasenlazadassimples-FabiRamiro/build (main)
$ cmake .. -G "MinGW Makefiles"
-- The CXX compiler identification is GNU 15.2.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/msys64/mingw64/bin/c++.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found Doxygen: C:/Strawberry/c/bin/doxygen.exe (found version "1.10.0") found components: doxygen missing components: dot
-- Configuring done (1.4s)
-- Generating done (0.0s)
-- Build files have been written to: C:/Users/fabi7/OneDrive/Documentos/Estructura de Datos/ds-u2-actividad-1-listasenlazadassimples-FabiRamiro/build
```

Figura 8: Configuración del proyecto con CMake

```
fabi7@fabian MINGW64 ~/OneDrive/Documentos/Estructura de Datos/ds-u2-actividad-1-listasenlazadassimples-FabiRamiro/build (main)
$ mingw32-make
[ 14%] Building CXX object CMakeFiles/SistemaIoTSensores.dir/src/main.cpp.obj
[ 28%] Building CXX object CMakeFiles/SistemaIoTSensores.dir/src/SensorBase.cpp.obj
[ 42%] Building CXX object CMakeFiles/SistemaIoTSensores.dir/src/SensorTemperatura.cpp.obj
[ 57%] Building CXX object CMakeFiles/SistemaIoTSensores.dir/src/SensorPresion.cpp.obj
[ 71%] Building CXX object CMakeFiles/SistemaIoTSensores.dir/src/Listageneral.cpp.obj
[ 85%] Building CXX object CMakeFiles/SistemaIoTSensores.dir/src/SerialReader.cpp.obj
[100%] Linking CXX executable SistemaIoTSensores.exe
[100%] Built target SistemaIoTSensores
```

Figura 9: Compilación exitosa con MinGW

5.2 ESP32 y Comunicación Serial



```
SimuladorSensores.ino
1  /**
2   * @file SimuladorSensores.ino
3   * @brief Simulador de sensores IoT para ESP32
4   * @author FabiRamiro
5   * @date 2025-10-31
6   *
7   * Este sketch simula el envío de datos de múltiples sensores
8   * (temperatura y presión) a través del puerto serial en formato:
9   * TIPO;ID;VALOR
10  *
11  * Ejemplos:
12  * - TEMP;T-001;23.5
13  * - PRES;P-105;85
14  */
15
16 // Intervalo de envío de datos (milisegundos)
17 const unsigned long INTERVALO_ENVIO = 2000;
18
19 // Variables para control de tiempo
20 unsigned long tiempoAnterior = 0;
21
22 // Contadores para IDs de sensores
23 int contadorTemp = 1;
24 int contadorPres = 1;
25
26 /**
27  * @brief Configuración inicial del ESP32
28  */
29 void setup()
30 {
31     // Inicializar comunicación serial a 115200 baudios
32     Serial.begin(115200);
33 }
```

```
Output
Writing at 0x000561e0 [=====] 100.0% 157863/157863 bytes...
Wrote 287200 bytes (157863 compressed) at 0x00010000 in 2.4 seconds (957.2 kbit/s).
Hash of data verified.
```

Figura 10: Sketch SimuladorSensores.ino en Arduino IDE

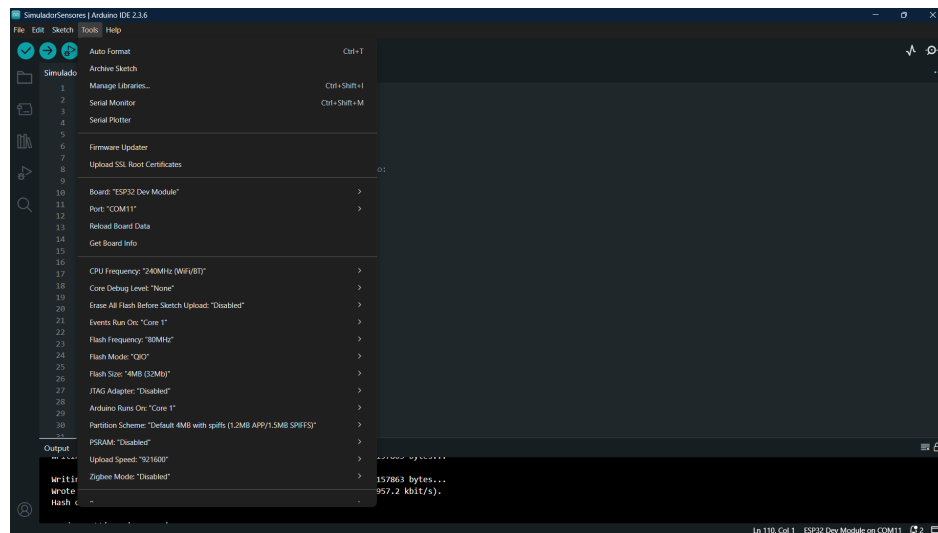


Figura 11: Configuración de placa ESP32 y puerto COM

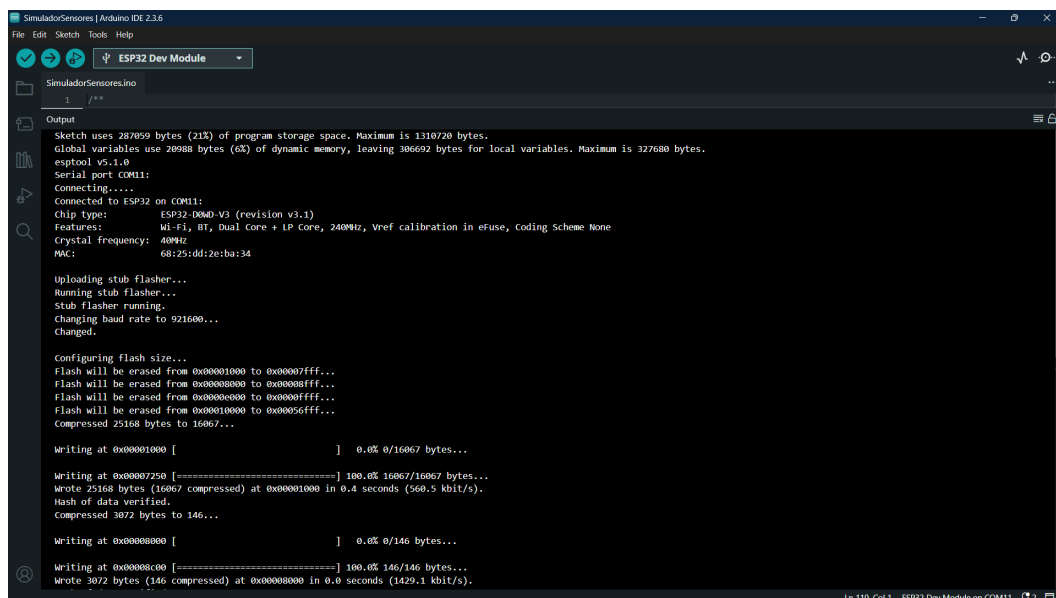


Figura 12: Carga exitosa del sketch al ESP32

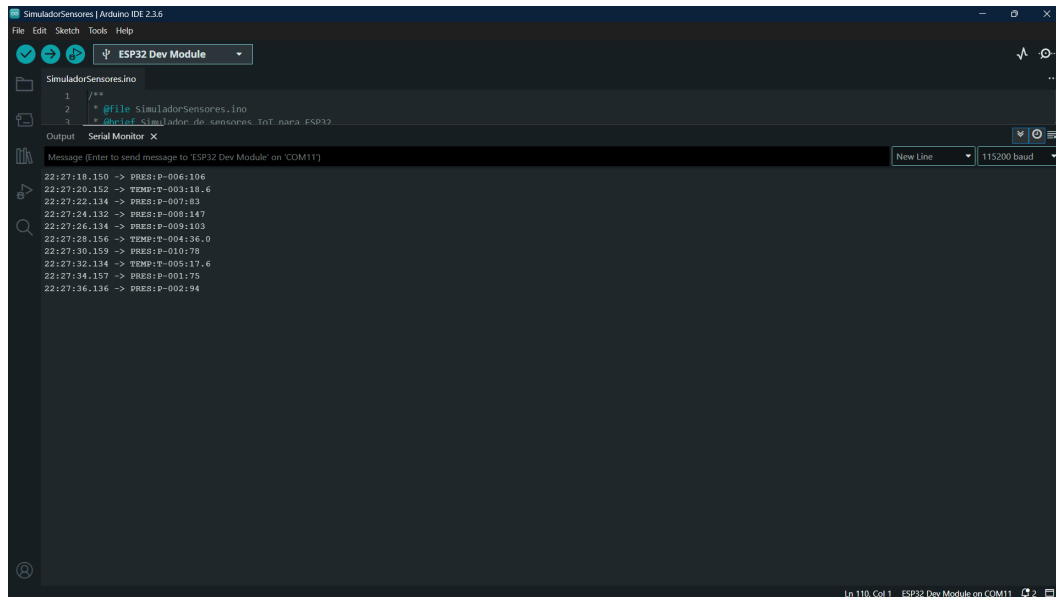


Figura 13: Serial Monitor mostrando datos simulados (115200 baud)

5.3 Ejecución del Programa C++

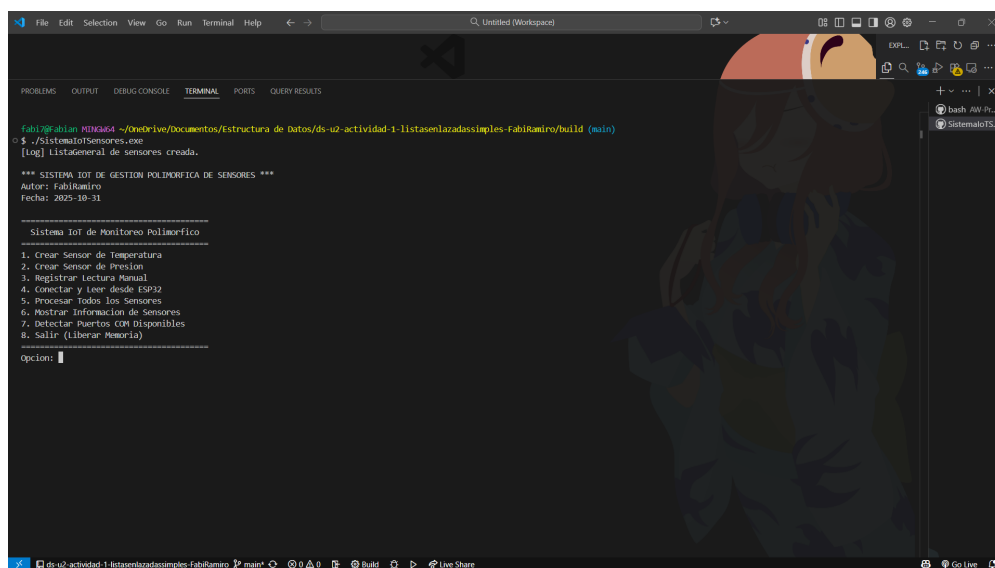


Figura 14: Menú principal del sistema

```
fabi@fabian MINGW64 ~/OneDrive/Documentos/Estructura de Datos/ds-u2-actividad-1-listasenlazadasimples-fabiRamiro/build (main)
$ ./SistemaIoTSensores.exe

1. Crear Sensor de Temperatura
2. Crear Sensor de Presion
3. Registrar Lectura Manual
4. Conectar y Leer desde ESP32
5. Procesar Todos los Sensores
6. Mostrar Informacion de Sensores
7. Detectar Puertos COM Disponibles
8. Salir (Liberar Memoria)

=====
Opcion: 7

=== Puertos COM Disponibles ===
[EN USO] COM11 (en uso por otra aplicacion)
=====

Presione Enter para continuar...

=====
Sistema IoT de Monitoreo Polimorfo
=====
1. Crear Sensor de Temperatura
2. Crear Sensor de Presion
3. Registrar Lectura Manual
4. Conectar y Leer desde ESP32
5. Procesar Todos los Sensores
6. Mostrar Informacion de Sensores
7. Detectar Puertos COM Disponibles
8. Salir (Liberar Memoria)

=====
Opcion: 7

=== Puertos COM Disponibles ===
[OK] COM11 (disponible)
=====

Presione Enter para continuar...
```

Figura 15: Detección automática de puertos COM disponibles

```
fabi@fabian MINGW64 ~/OneDrive/Documentos/Estructura de Datos/ds-u2-actividad-1-listasenlazadasimples-fabiRamiro/build (main)
$ ./SistemaIoTSensores.exe

Sistema IoT de Monitoreo Polimorfo

1. Crear Sensor de Temperatura
2. Crear Sensor de Presion
3. Registrar Lectura Manual
4. Conectar y Leer desde ESP32
5. Procesar Todos los Sensores
6. Mostrar Informacion de Sensores
7. Detectar Puertos COM Disponibles
8. Salir (Liberar Memoria)

=====
Opcion: 4

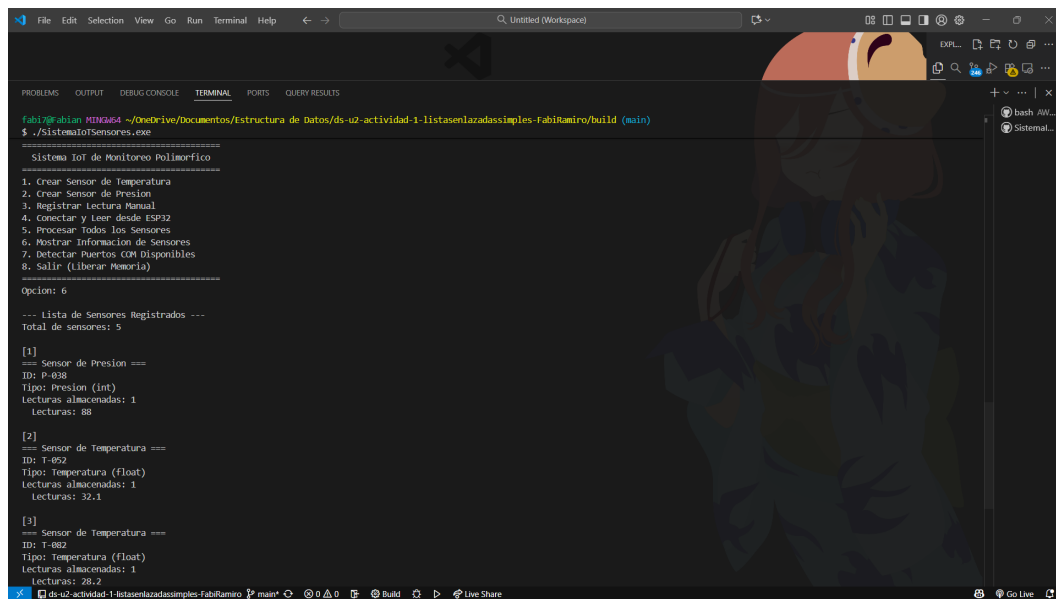
Ingrese el puerto COM (ej: COM3): COM11
[SerialReader] Intentando conectar a COM11...
[SerialReader] Conectado exitosamente.

Cuántas lecturas desea capturar? (0 = continuo): 5

--- Capturando datos del ESP32 ---

[ESP32] Recibido: PRES:P-038:88
[Log] ListaSensorCT> creada.
[Log] SensorPresion 'P-038' creado.
[Log] Sensor 'P-038' insertado en la lista de gestion.
[Log] NodoCT> insertado. Valor: 88
[P-038] Lectura registrada: 88 PSI
[ESP32] Recibido: TEMP:T-052:32.1
[Log] ListaSensorCT> creada.
[Log] SensorTemperatura 'T-052' creado.
[Log] Sensor 'T-052' insertado en la lista de gestion.
[Log] NodoCT> insertado. Valor: 32.1
[T-052] Lectura registrada: 32.1 °C
[ESP32] Recibido: TEMP:T-082:28.2
[Log] ListaSensorCT> creada.
[Log] SensorTemperatura 'T-082' creado.
[Log] Sensor 'T-082' insertado en la lista de gestion.
```

Figura 16: Captura de datos en tiempo real desde ESP32



```
fabi@fabian: MINGW64 ~/OneDrive/Documentos/Estructura de Datos/ds-u2-actividad-1-listasenlazadassimples-fabiRamiro/build (main)
$ ./SistemaIoTSensores.exe

=====
Sistema IoT de Monitoreo Polimorfico
=====
1. Crear Sensor de Temperatura
2. Crear Sensor de Presion
3. Registrar Lectura Manual
4. Conectar y Leer desde ESP32
5. Procesar Todos los Sensores
6. Mostrar Informacion de Sensores
7. Detectar Puertos COM Disponibles
8. Salir (Liberar Memoria)
=====
Opcion: 6

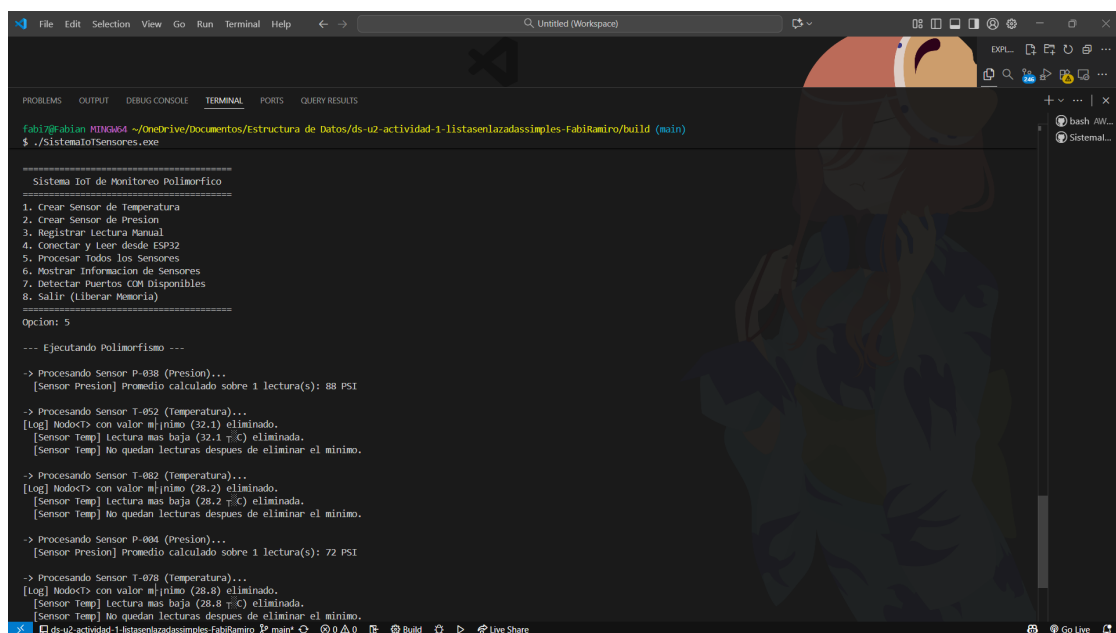
--- Lista de Sensores Registrados ---
Total de sensores: 3

[1]
=== Sensor de Presion ===
ID: P-038
Tipo: Presion (int)
Lecturas almacenadas: 1
Lecturas: 88

[2]
=== Sensor de Temperatura ===
ID: T-052
Tipo: Temperatura (float)
Lecturas almacenadas: 1
Lecturas: 32.1

[3]
=== Sensor de Temperatura ===
ID: T-082
Tipo: Temperatura (float)
Lecturas almacenadas: 1
Lecturas: 28.2
```

Figura 17: Información detallada de sensores registrados



```
fabi@fabian: MINGW64 ~/OneDrive/Documentos/Estructura de Datos/ds-u2-actividad-1-listasenlazadassimples-fabiRamiro/build (main)
$ ./SistemaIoTSensores.exe

=====
Sistema IoT de Monitoreo Polimorfico
=====
1. Crear Sensor de Temperatura
2. Crear Sensor de Presion
3. Registrar Lectura Manual
4. Conectar y Leer desde ESP32
5. Procesar Todos los Sensores
6. Mostrar Informacion de Sensores
7. Detectar Puertos COM Disponibles
8. Salir (Liberar Memoria)
=====
Opcion: 5

--- Ejecutando Polimorfismo ---

-> Procesando Sensor P-038 (Presion)...
[Sensor Presion] Promedio calculado sobre 1 lectura(s): 88 PSI

-> Procesando Sensor T-052 (Temperatura)...
[Log] Nodo<T> con valor m[inimo] (32.1) eliminado.
[Sensor Temp] Lectura mas baja (32.1 °C) eliminada.
[Sensor Temp] No quedan lecturas despues de eliminar el minimo.

-> Procesando Sensor T-082 (Temperatura)...
[Log] Nodo<T> con valor m[inimo] (28.2) eliminado.
[Sensor Temp] Lectura mas baja (28.2 °C) eliminada.
[Sensor Temp] No quedan lecturas despues de eliminar el minimo.

-> Procesando Sensor P-084 (Presion)...
[Sensor Presion] Promedio calculado sobre 1 lectura(s): 72 PSI

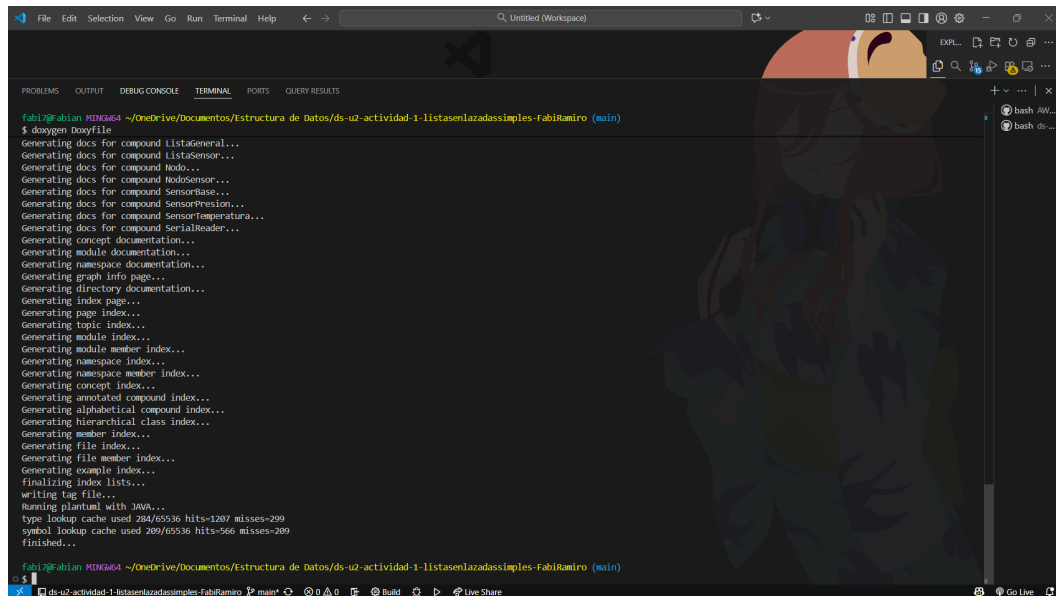
-> Procesando Sensor T-078 (Temperatura)...
[Log] Nodo<T> con valor m[inimo] (28.8) eliminado.
[Sensor Temp] Lectura mas baja (28.8 °C) eliminada.
[Sensor Temp] No quedan lecturas despues de eliminar el minimo.
```

Figura 18: Procesamiento polimórfico de sensores



Figura 19: Liberación de memoria en cascada al salir

5.4 Documentación Doxygen



```
fabir@abian MINGW64 ~/OneDrive/documentos/Estructura de Datos/ds-u2-actividad-1-listasenlazadassimples-FabiRamiro (main)
$ doxygen doxyfile
Generating docs for compound ListaGeneral...
Generating docs for compound ListaSensor...
Generating docs for compound Nodo...
Generating docs for compound NodoSensor...
Generating docs for compound SensorBase...
Generating docs for compound SensorPresion...
Generating docs for compound SensorTemperatura...
Generating docs for compound SerialReader...
Generating concept documentation...
Generating module documentation...
Generating namespace documentation...
Generating graph info page...
Generating directory documentation...
Generating index page...
Generating page index...
Generating topic index...
Generating module index...
Generating module member index...
Generating namespace index...
Generating namespace member index...
Generating concept index...
Generating annotated compound index...
Generating alphabetical compound index...
Generating hierarchical class index...
Generating member index...
Generating file index...
Generating file member index...
Generating example index...
finalizing index lists...
writing tag files...
Running plantuml with JAVA...
type lookup cache used 284/6536 hits=1207 misses=299
symbol lookup cache used 209/6536 hits=566 misses=209
finished...

fabir@abian MINGW64 ~/OneDrive/documentos/Estructura de Datos/ds-u2-actividad-1-listasenlazadassimples-FabiRamiro (main)
```

Figura 20: Proceso de generación de documentación con Doxygen

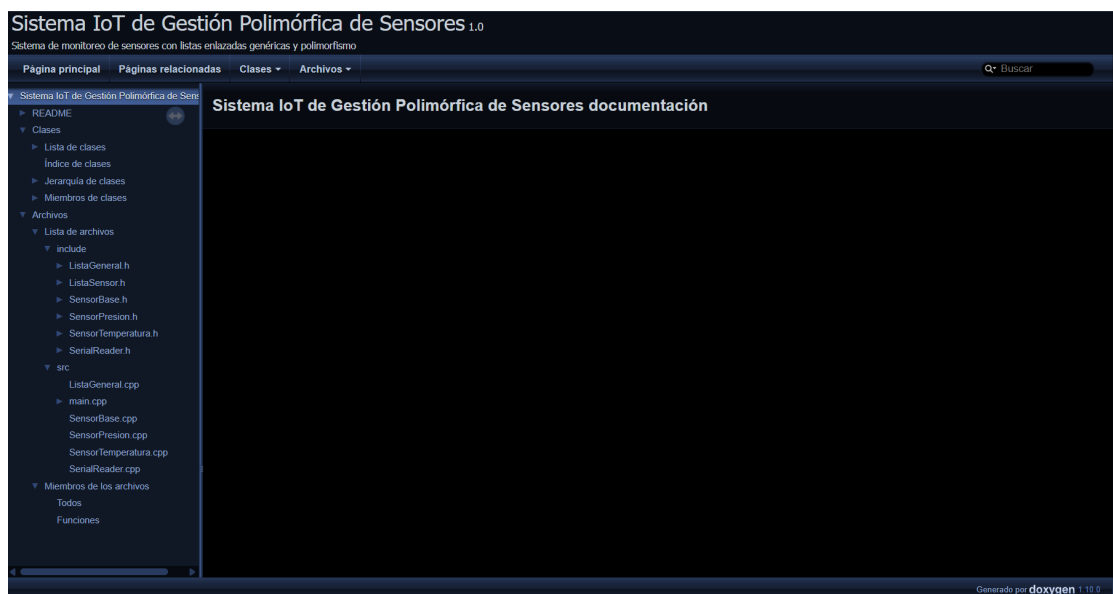


Figura 21: Página principal de la documentación HTML

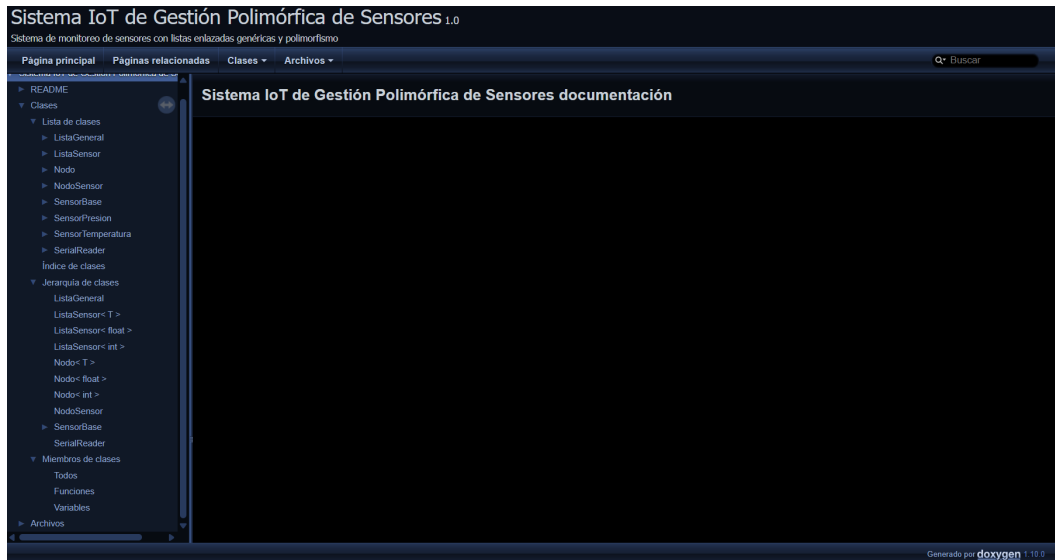


Figura 22: Lista de clases documentadas

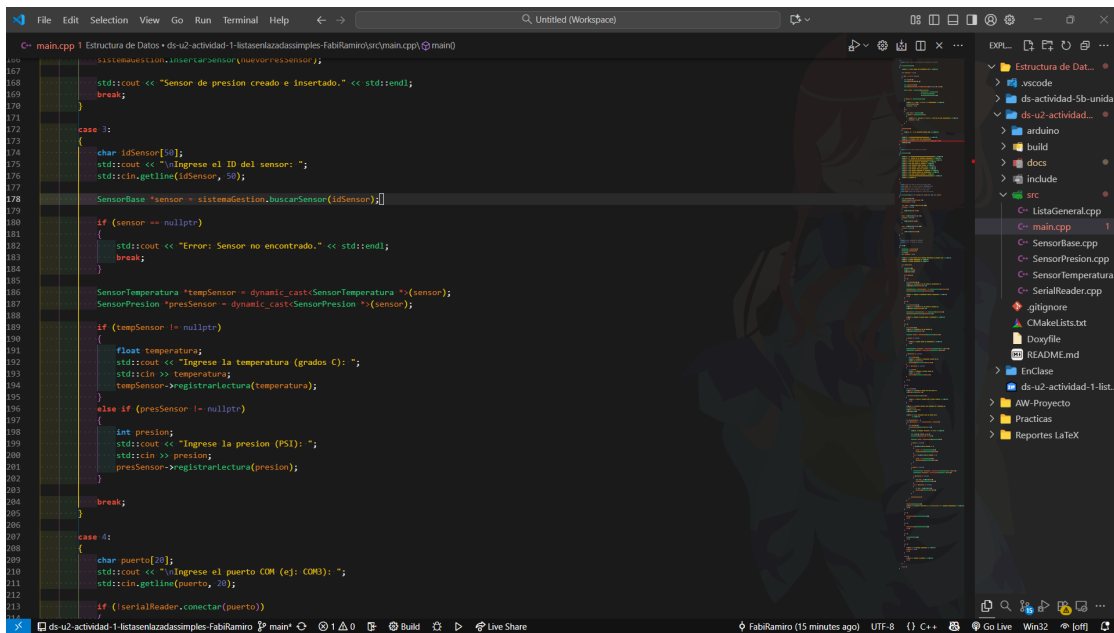


Figura 23: Ejemplo de código con uso polimórfico

6 Conclusiones

6.1 Logros Alcanzados

El proyecto cumplió satisfactoriamente con todos los objetivos planteados:

- **Jerarquía polimórfica:** Se implementó una clase base abstracta (`SensorBase`) con métodos virtuales puros, garantizando que las clases derivadas implementen el comportamiento requerido.
- **Listas enlazadas genéricas:** Se desarrolló una implementación completa de lista enlazada simple con templates (`ListaSensor<T>`), funcional para tipos `int` y `float`.

- **Gestión manual de memoria:** Se aplicó correctamente la Regla de los Tres, implementando destructores, constructores de copia y operadores de asignación. No se detectaron fugas de memoria.
- **Integración con hardware:** La comunicación serial con ESP32 funciona establemente a 115200 baud, permitiendo captura de datos en tiempo real.
- **Documentación completa:** Todo el código está documentado con comentarios Doxygen, generando documentación HTML navegable y profesional.
- **Sistema redistribuible:** El uso de CMake permite compilar el proyecto en diferentes entornos con facilidad.

6.2 Conceptos Aplicados Exitosamente

6.2.1. Programación Orientada a Objetos

- Abstracción mediante clases abstractas
- Encapsulamiento de datos y comportamiento
- Herencia para reutilización de código
- Polimorfismo en tiempo de ejecución
- Destructores virtuales para correcta liberación

6.2.2. Estructuras de Datos

- Implementación manual de listas enlazadas
- Gestión de punteros y nodos
- Operaciones fundamentales: inserción, búsqueda, eliminación
- Algoritmos de cálculo (promedio, mínimo)

6.2.3. Programación Genérica

- Templates de clases
- Código independiente del tipo
- Instanciación implícita de templates

6.3 Dificultades Encontradas y Soluciones

Tabla 6: Problemas y soluciones durante el desarrollo

Problema	Solución Aplicada
Conflicto con OneDrive en generación de documentación	Uso de rutas absolutas en Doxyfile
Caracteres basura en Serial Monitor	Configuración correcta del baud rate a 115200
Archivo Doxyfile.bak editado por error	Renombrar el archivo correcto y eliminar .bak
Múltiples definiciones en compilación	Verificar correspondencia correcta entre headers y .cpp
Documentación Doxygen vacía	Configurar correctamente INPUT y OUTPUT_DIRECTORY

6.4 Aprendizajes Significativos

1. **Importancia de destructores virtuales:** Sin ellos, la liberación polimórfica de memoria causa fugas en las clases derivadas.
2. **Templates en headers:** A diferencia de las clases normales, los templates deben implementarse completamente en archivos de encabezado (.h) debido a limitaciones del modelo de compilación de C++.
3. **Windows API para comunicación serial:** No se requieren bibliotecas externas para comunicación serial en Windows; la API nativa (`CreateFile`, `ReadFile`, etc.) es suficiente y eficiente.
4. **CMake como sistema de construcción:** Facilita la portabilidad del proyecto entre diferentes compiladores y sistemas operativos.
5. **Doxygen para documentación profesional:** El uso de comentarios especiales genera automáticamente documentación navegable sin esfuerzo adicional.
6. **Downcasting seguro:** El uso de `dynamic_cast<>` permite acceder a métodos específicos de clases derivadas de forma segura.

6.5 Posibles Extensiones Futuras

El sistema diseñado es extensible y podría mejorarse con:

- Agregar más tipos de sensores (humedad, vibración, luz)
- Implementar persistencia de datos en archivos o base de datos
- Desarrollar interfaz gráfica (Qt o similar)
- Soporte para múltiples ESP32 simultáneos

- Análisis estadístico avanzado (desviación estándar, percentiles)
- Sistema de alertas basado en umbrales
- Gráficas en tiempo real de lecturas
- Servidor web para monitoreo remoto

6.6 Reflexión Final

Este proyecto integra de manera exitosa conceptos fundamentales de programación en C++, estructuras de datos y sistemas embebidos. La prohibición de usar STL, aunque desafiante, permitió comprender profundamente:

- Cómo funcionan las estructuras de datos internamente
- La importancia de la gestión manual de memoria
- Los peligros de las fugas de memoria y cómo evitarlas
- El verdadero poder del polimorfismo y los templates

El sistema resultante es funcional, eficiente, documentado y extensible, cumpliendo con todos los requisitos académicos y demostrando dominio de programación de sistemas de bajo nivel.