Recuperatorio Segundo Parcial

Ejercicio 1 : Implementación de Vector Genérico

Completar la implementación de una clase Vector genérica con funcionalidades avanzadas de serialización y operaciones vectoriales.

Archivos Proporcionados

Se entregan los siguientes archivos parcialmente implementados:

1. Vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

#include <stdexcept>
#include <cmath>
#include <iostream>
#include <fstream>

template <typename T>
class Vector {
private:
    T* componentes;
    int dimension;

public:
    // Constructores
    Vector();
    Vector(const T* arr, int dim);
    Vector(const Vector& other);

    // Destructor
    ~Vector();

    // Operadores
    Vector operator+(const Vector& other) const;
    Vector operator-(const Vector& other) const;
    T operator*(const Vector& other) const;
    Vector operator%(const Vector& other) const;
```

```
    Vector operator*(T scalar) const;
    T& operator[](int index);
    bool operator==(const Vector& other) const;

    // Métodos adicionales
    T norma() const;
    void normalizar();
    int getDimension() const { return dimension; }

    // Sobrecarga de operadores de flujo
    template <typename U>
    friend std::ostream& operator<<(std::ostream& os, const Vector<U>& v);

    template <typename U>
    friend std::istream& operator>>(std::istream& is, Vector<U>& v);

    // Métodos de serialización
    void guardar(const std::string& archivo) const;
    void cargar(const std::string& archivo);
};

#endif // VECTOR_H
```

2. Vector.cpp

```
#include "Vector.h"
#include <stdexcept>
#include <cmath>

// TODO: Implementar los métodos del Vector

// Implementación del producto cruz (PARCIALMENTE IMPLEMENTADA)
template <typename T>
Vector<T> Vector<T>::operator%(const Vector& other) const {
    if (dimension != 3 || other.dimension != 3) {
        // TODO: Completar manejo de excepción
    }

    // TODO: Calcular componentes del producto cruz
}
```

3. main.cpp

```
#include <iostream>
#include "Vector.h"

int main() {
    // TODO: Implementar casos de prueba

    return 0;
}
```

Tareas a Realizar

1. Implementar **Vector.cpp**:
   - Completar los constructores
   - Implementar operadores aritméticos (`+`, `-`, `*`)
   - Finalizar la implementación del producto cruz
   - Añadir método de normalización

2. Desarrollar **main.cpp**:
   - Crear casos de prueba para:
     * Creación de vectores de diferentes tipos
     * Operaciones vectoriales
     * Manejo de excepciones
     * Operaciones de normalización

Requerimientos Adicionales

- Manejar correctamente la memoria dinámica
- Implementar control de errores con excepciones
- Demostrar el uso de plantillas (templates)
- El código debe compilar sin errores
- Verificar la funcionalidad con vectores de enteros y punto flotante

Criterios de Evaluación

- Correctitud de la implementación
- Manejo adecuado de memoria
- Control de excepciones
- Claridad y legibilidad del código
- Casos de prueba completos
- Conceptos teóricos de forma oral

Entrega

- Código fuente completo
- Ejecutable generado
- Informe breve explicando decisiones de diseño

Consideraciones Especiales

- Prohibido usar bibliotecas externas no estándar
- Se permite el uso de herramientas basadas en IA
- No se permite compartir código entre estudiantes

Ejercicio 2 : Procesamiento de figuras geométricas

Se proporciona una implementación inicial de una jerarquía de clases para representar figuras geométricas y un functor **FiltrarPorTipo** para filtrar un vector de figuras.

figuras.h

class Figura { public: virtual double calcularArea() const = 0; virtual double calcularPerimetro() const = 0; // ... otros métodos virtuales puros };

Circulo.h

class Circulo : public Figura { // ... atributos y métodos };

Rectangulo.h

class Rectangulo : public Figura { // ... atributos y métodos };

Triangulo.h

class Triangulo : public Figura { // ... atributos y métodos };

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
// ... clases Figura, Circulo, Rectangulo, Triangulo ...

int main() {
        std::vector<Figura*> figuras; // ... crear figuras y agregarlas al vector // Ordenar por
        área
        std::sort(figuras.begin(), figuras.end(), OrdenarPorArea()); // Filtrar círculos

        auto it = std::stable_partition(figuras.begin(), figuras.end(), FiltrarPorTipo()); //
        Imprimir figuras filtradas (círculos)

        for (auto figura : std::vector<Figura*>(figuras.begin(), it)) { std::cout << "Área: " <<
        figura->calcularArea() << std::endl; }
}
```

Tareas:

Documentación:

- Documentar el código: Añadir comentarios claros y concisos a todas las clases, métodos y funciones, explicando su propósito y funcionamiento.
- Crear un diagrama de clases: Representar gráficamente la jerarquía de clases, mostrando las relaciones de herencia, atributos y métodos.
- Explica el uso del functor: Describir el papel del functor **FiltrarPorTipo** en el proceso de filtrado y cómo se utiliza con la STL.

Modificaciones y Ampliaciones:

- Personalización del functor: Modificar el functor **FiltrarPorTipo** para que sea más genérico, utilizando un template y conceptos para permitir filtrar por cualquier tipo de figura derivada de Figura.
- Nuevos criterios de filtrado: Implementa nuevos functores para filtrar las figuras según criterios como área, perímetro o color (si se agrega un atributo de color a las figuras).
- Ordenamiento personalizado: Crea un functor para ordenar las figuras según un criterio específico (por ejemplo, por área de mayor a menor).

Pruebas:

Escribir casos de prueba: Diseñar casos de prueba para verificar que las modificaciones funcionen correctamente.

Evaluación:

Se evaluará:

- Correctitud de la implementación
- La claridad y precisión de la documentación
- La corrección del diagrama de clases
- La comprensión de los conceptos de herencia, polimorfismo y programación genérica
- La capacidad de crear código bien estructurado y fácil de mantener
- Conceptos teóricos de forma oral