**Instituto Tecnológico de Costa Rica**

**Proyecto 1: MPointers**

Estudiante Bryan Monge Navarro – 2023026192

Profesor José Isaac Ramírez Herrera

Curso Algoritmos y Estructura de Datos II

Fecha: II Semestre 2024

Link al repositorio de GitHub: https://github.com/JourneyBry05/TEC/tree/main/II\_Semester2024/AlgorithmsAndDataStructureII/ProjectOne

**Introducción**

Como parte del proyecto se ha pedido como requerimientos el diseño e implementación de un nuevo tipo de dato en el lenguaje de programación C++ denominado MPointer. Consiste en crear una biblioteca para MPointer que pueda ser utilizada por cualquier programador para el desarrollo de una aplicación en dicho lenguaje. MPointer es una clase template (MPointer<T>) que encapsula un puntero.

Antes de usar MPointer, el programador debe inicializar la clase singleton MPointerGC que se ejecuta como un thread cada n segundo. Esta clase tiene una lista enlazada donde guarda direcciones de memoria de los MPointer conocidos, dándole a MPointer, un ID autogenerado.

El destructor de MPointer llama a MPointerGC para indicar que la referencia se ha destruido. Una vez que el conteo de referencias de un MPointer llegue a cero, el MPointerGC lo libera, evitando memory leaks

Para probar la funcionalidad de MPointer, se deberá implementar QuickSort, BubbleSort e InsertionSort utilizando listas doblemente enlazadas que utilizan MPointers internamente. Adicional a lo anterior, se deberá incluir pruebas unitarias con GTest con una cobertura del 100% del código

Tabla de contenido

[Descripción del problema 4](#_Toc177163370)

[Descripción de la solución 5](#_Toc177163371)

[Creación de un tipo de dato MPointer: 5](#_Toc177163372)

[Creación de GC 5](#_Toc177163373)

[Implementación de algoritmos de ordenamiento: 6](#_Toc177163374)

[Ejemplo de compilación 7](#_Toc177163375)

[Diagrama UML 7](#_Toc177163376)

# Descripción del problema

Mediante la implementación de una librería que permita administrar punteros de manera eficiente, y que además implemente un recolector de basura que libere automáticamente la memoria no utilizada permite evitar problemas comunes como memory leaks y se maneja eficientemente el uso de punteros.

El mal uso de punteros generan errores difíciles de depurar, que cuando se trabaja en proyectos de gran escala o recursos limitados, puede generar desperdicios de recursos y asegurar el buen funcionamiento del software.

El proyecto también requiere la implementación de algoritmos de ordenamiento (como QuickSort, BubbleSort e InsertionSort) utilizando estructuras de datos que internamente gestionan la memoria a través de MPointer, sin que los algoritmos sean conscientes de su existencia.

Además, se incluye la implementación de un recolector de basura (MPointerGC), que se encargará de liberar automáticamente la memoria que ya no se está utilizando, evitando fugas de memoria

# Descripción de la solución

Creación de un tipo de dato MPointer:Fue diseñado como una plantilla que encapsula un puntero del tipo T. Esto para permitir gestionar automáticamente la memoria asignada para objetos primitivos; así, cada instancia de MPointer crea un nuevo puntero dinámico para asignar memoria

El acceso a los valores apuntados se realiza mediante la sobrecarga de los operadores “\*” y “&”, igualmente para “=” para la copia de punteros de forma segura.

La clase se integra con MPointerGC para realizar la gestión automática de memoria.

La posibilidad de tomar otra alternativa para la creación de un nuevo tipo de dato es muy baja debido a que la única manera de proceder es mediante la clase *template*. Se ha considerado el uso de punteros inteligentes como std::shared\_ptr y std::unique\_ptr, pero se prefirió implementarlos de manera personalizada con MPointer.

En cuanto a limitaciones, MPointer no es compatible con otro tipo de punteros inteligentes ni variables personalizables por el desarrollador

Durante el desarrollo del proyecto se tuvo problemas creando el GC, para que este pudiera manejar las referencias creadas y destruidas, esto es, mantener un conteo de referencias y eliminarlas cuando lleguen a 0. Fue difícil durante la solución la implementación de los algoritmos de ordenamiento, para que estos no conocieran la existencia de MPointer; fue relativamente sencilla su solución debido a que se debía simplemente cambiar el tipo de dato desde un código de algoritmos de ordenamiento encontrado en Internet.

En la etapa de optimización, se encontró que los valores almacenados en memoria con MPointer eran incompatibles con los algoritmos de ordenamiento

## Creación de GC

El recolector de referencias MPointerGC fue diseñado como una clase singleton que se encarga de gestionar la memoria asignada a los MPointer. Este debe realizar un seguimiento de las referencias activas, y cuando el conteo de referencias de un puntero llegue a cero, MPointerGC libera automáticamente la memoria correspondiente para prevenir fugas de memoria.

La recolección de basura manual y automática se realiza a partir de métodos para incrementar y decrementar el conteo de referencias, acompañado de un método para ejecutar la recolección de basura *collectGarbage() que libera la memoria de los punteros cuya referencia es cero.*

Como alternativa se pensó crear un recolector de basura utilizando un hilo separado, pero requería mayor complejidad, creando problemas al querer testear el código en caso de errores frecuentes.

Se ha considerado el uso de punteros inteligentes como std::shared\_ptr y std::unique\_ptr, pero se prefirió implementarlos de manera personalizada con MPointer, permitiendo personalizar el comportamiento según las necesidades del proyecto.

Ciertas limitaciones pueden deberse a que la recolección se realiza de manera manual, es decir que si no se llama el recolector en momentos adecuados este puede generar fugas de memoria. También se puede llevar a un memory leak cuando dos Pointer se refieren entre sí, el GC no podrá recolectar la memoria.

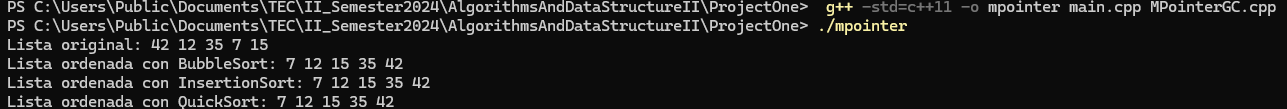
Durante la implementación, se encontró con una serie de errores en la disminución en el conteo de referencias, ya que inicialmente se encontró que algunos punteros seguían manteniendo referencias después de ser liberados

Implementación de algoritmos de ordenamiento:Estas estructuras de datos fueron implementadas para trabajar con el nuevo tipo de dato MPointer, asegurándose que los algoritmos no fueran conscientes de la existencia de MPointer. Las implementaciones establecidas se basan en el básico script que se encuentra en internet, simplemente cambia el valor del tipo de dato, ya que esto lo gestiona MPointer

Una alternativa que hubiera simplificado la implementación, era cambiar los algoritmos sobre punteros crudos, pero hubiera requerido que se gestionaran manualmente la memoria, incrementando el riesgo de fuga de memoria

Durante las primeras fases de testing de la implementación, los algoritmos estaban intercambiando punteros y corrompiendo el valor almacenado en la dirección de memoria. También hubo casos en los que se intentó acceder a la memoria que ya había sido liberada, lo que causaba errores de segmentación

# Ejemplo de compilación



# Diagrama UML

