ITCR

Ingeniería en Computadores

Algoritmos y Estructuras de Datos 2

Documentación Proyecto 1

Estudiantes:
Esteban Campos Abarca - 2022207705
Jafet Díaz Morales - 2023053249

I Semestre 2025

Tabla de Contenidos

Introducción

En este documento se presenta la implementación de clases que encapsulan el uso de punteros en C + + con el fin de aplicar conceptos de manejo de memoria, investigar y desarrollar una aplicación en el lenguaje de programación C + +, además de investigar sobre POO y patrones de diseño en C + +, pudiendo así tener buenas prácticas de programación (documentar, uso de UML, etc.)

Breve Descripción del Problema

El proyecto consiste en dos componentes principales: el administrador de memoria (Memory Manager) y la biblioteca MPointers.

El administrador de memoria reserva un bloque de memoria de cierto tamaño y lo administra. La biblioteca MPointers permite a las aplicaciones que lo usen, interactuar con el administrador de memoria y el bloque de memoria reservado por este.

El Memory Manager es un servicio que escucha comandos mediante GRPC (utilizando una biblioteca existente para este fin) para administrar un bloque de memoria. Este comando indicará el "Listen_port", que es el puerto donde se escuchan las peticiones mediante GRPC, el tamaño en megabytes de la memoria que será administrada ("Size_MB") y una carpeta ("DUMP_FOLDER") en la que después de cada petición que modifique la memoria, se crea un archivo que muestra el estado de la memoria.

En general, Memory Manager sigue el siguiente proceso:

1. Reserva la memoria en el heap a través de la única asignación de memoria que puede haber en todo el proyecto.
2. Inicia el servidor GRPC.
3. Espera las peticiones de los clientes y las procesa al recibirlas.
4. Crear el archivo en el folder dump después de cada modificación.

Las peticiones que escucha memory manager son:

1. Create (size, type): crea un espacio en la memoria para el tamaño y tipo de datos indicado en la petición. Retorna un Id que identifica el espacio en memoria recién creado.
2. Set(id, value): guarda un valor determinado en la posición de memoria indicado por Id.

3. Get(id): retorna el valor guardado en el bloque de memoria identificado por Id.
4. IncreaseRefCount(id): incrementa el conteo de referencias para el bloque indicado por Id.
5. DecreaseRefCount(Id): decrementa el conteo de referencias para el bloque indicado por Id.

Además, Memory manager cuenta con un garbage collector que lleva el control de las referencias a memoria (qué memoria está libre y cuál no).

Por su lado, MPointers es una clase template (MPointer) que permite interactuar con Memory Manager mediante peticiones GRPC. Se utiliza por el programa cliente. Sobrecarga operadores como &, * entre otros, para comportarse como un pointer. Sin embargo, toda la memoria se mantiene remotamente en Memory Manager.

MPointer tiene un método estático llamado Init que recibe el puerto en el que memory manager escucha. Este método debe invocarse al inicio del programa cliente. A través de la sobrecarga, al comunicarse mediante GRPC, MPointers puede asignar y acceder a espacios de memoria del Memory Manager

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

Descripción de la solución

Se procederá a explicar la solución en referente a cada requerimiento. Los requerimientos 0-4 son de Memory Manager, los requerimientos 5-8 son de MPointers

Requerimiento 0: Línea de comandos con los parámetros especificados

Es un comando custom que recibe la main. Su función es iniciar el servidor en el puerto escogido y con la memoria por reservar deseada.

Requerimiento 1: Comunicación mediante GRPC

Desarrollar la comunicación gRPC consta de dos partes: construir un archivo .proto que define los servicios con varios métodos tanto del lado del cliente como del servidor al compilarse con protobuf. Las definiciones de los métodos son ocultas para el cliente, pero las reconoce por medio de los archivos creados para este. Por otro lado, la conexión servidor cliente en el momento de ejecutar el programa funciona por medio de gRPC.

Se intentó este procedimiento sin éxito por incompatibilidad de versiones entre las instalaciones que no se lograron corregir. Se implementó el mismo tipo de comunicación pero utilizando un servidor-cliente mediante sockets, pero el funcionamiento es muy similar en sí al de gRPC.

Requerimiento 2: Implementación de los cinco tipos de peticiones

Antes de las peticiones se inicia el servidor. Este reserva una cantidad de memoria escogida por el usuario en bytes desde una clase Heap. Esto funciona por medio de una clase Word, la cual representa un espacio de memoria de 8 bytes más metadatos como id (dirección de memoria) y conteo de referencias. La memoria se reserva al hacer un malloc del tamaño de Word multiplicado por la cantidad size. El espacio reservado se conoce como memory_space
Create(size, type): reserva un espacio dentro de memory_space de tamaño size. Esto se logra aumentando el conteo de referencias de una sección libre en la memoria. El índice de esta sección libre es devuelto por Create. El tipo de dato es manejado principalmente por el usuario al declararlo en el MPointer.
Set(id, value): recibe un valor como array de bytes (std::vector<uint8_t>) y guarda cada byte individual en Words consecutivas de memory_space según la dirección dada por id. Si hay una discrepancia entre los tamaños de los datos, toma el tamaño más corto.
Get(id): retorna un array de bytes según la dirección dada por id.
IncreaseRefCount(id) y DecreaseRefCount(id): ambos están contemplados en ModifyRefCount(id, value). Esta función añade value al ref_count de la dirección de memoria especificada. Value es 1 o -1 y esto se determina en una función privada aparte del MemoryManager que reconoce la petición del cliente de ya sea crear o deshacer un puntero.

Requerimiento 3: Implementación del Garbage Collector

Una variable entera de MemoryManager detecta cuando suficientes transacciones se han efectuado y se debe usar el garbage collector. Esto con el fin de no usarlo ni manualmente ni por cada transacción de memoria. Funciona al recorrer la memoria y deshacer cambios en Words que no estén ocupadas (ref_count = 0);

Sin embargo, daba errores y no se pudo implementar.

Requerimiento 4: Implementación de la defragmentación de memoria

No se implementó.

Requerimientos 5, 6 y 7: Comunicación GRPC con Memory Manager y sobrecarga de operadores

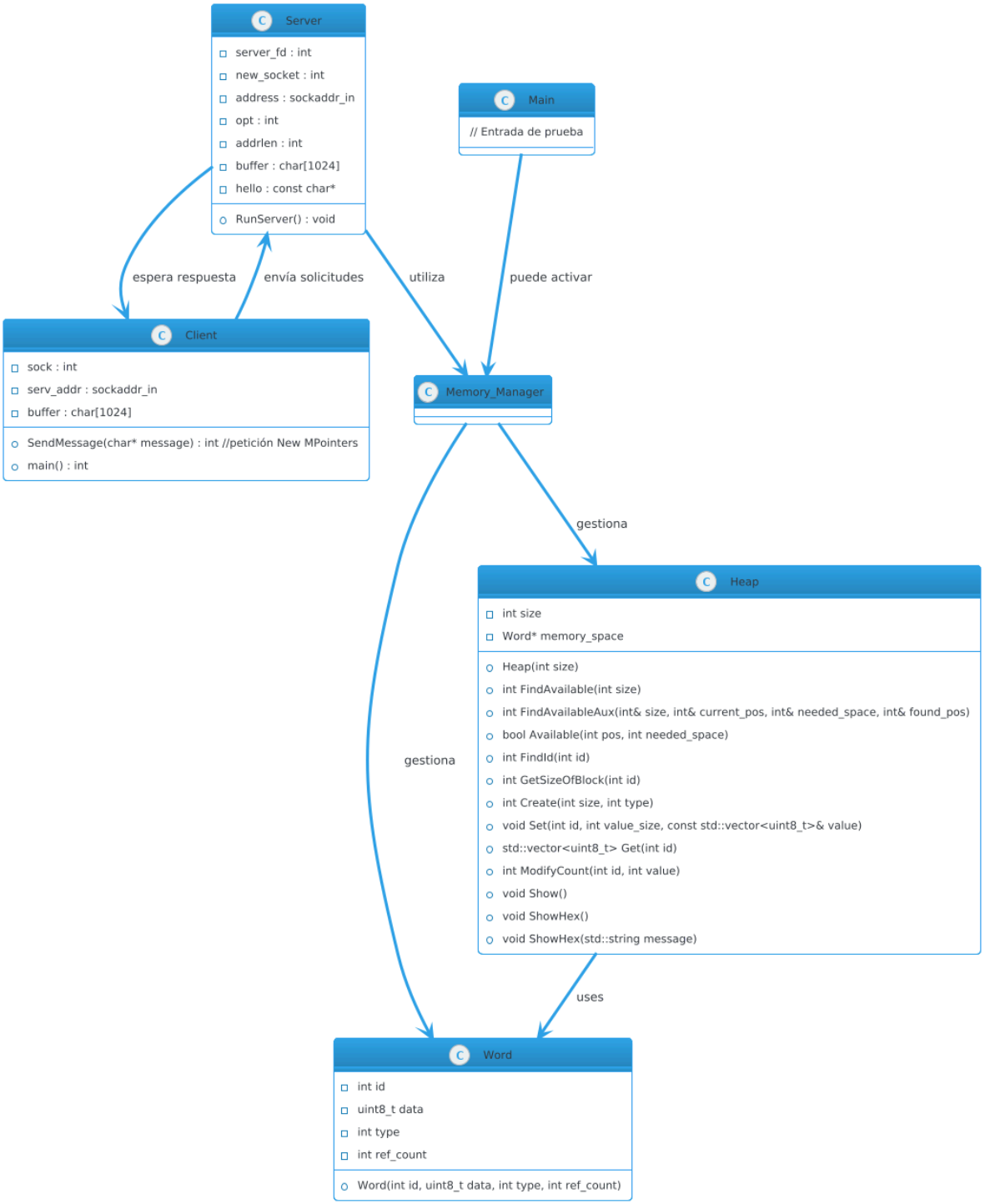
Los MPointer del lado del cliente se comunican con MemoryManager desde su definición. El constructor y la sobrecarga de los operadores = y * resultan en llamadas a MemoryManager. Al asignarse la dirección de un puntero a otro, se aumenta el conteo de referencias como si se hubiera creado uno nuevo con el constructor. Al asignarse un valor al objeto al que apunta el puntero (*ptr = value), se llama la función Set. Al asignarse el valor del puntero a una variable (var = *ptr), se llama a Get(). El destructor llama a ModifyRefCount para disminuir ref_count en el id del puntero. El puntero guarda un id y MemoryManager es responsable de encontrar las direcciones absolutas del memory_space usando este id.

Requerimiento 8: Pruebas con listas enlazadas

Se crea una clase Nodo<T> que contiene MPointer a otro nodo y un dato <T>. Una clase Lista<T> se encarga de conectar varios nodos por medio de un método Add. Además Lista tiene otros métodos para manejo de Listas

Diagramas UML

A continuación se muestran los diagramas UML con las clases más importantes del proyecto.



Enlace a github:
https://github.com/EstebanACamposA/campos-diaz-mpointers/tree/main