Algoritmos y Estructuras de Datos II

Trabajo Práctico N° 6	Unidad 6
Modalidad: Semi -Presencial	**Estratégica Didáctica:** Trabajo individual
Metodología de Desarrollo: Det. docente	**Metodología de Corrección:** Via Classroom.
Carácter de Trabajo: Obligatorio – Con Nota	**Fecha Entrega:** A confirmar por el Docente.

MARCO TEÓRICO (Cuestionario reflexivo):

Unidad 6 - Trabajo Práctico – Memory Leaks

1. ¿Qué entiende por memory leak?

Un memory leak (pérdida de memoria) es una condición que ocurre cuando una aplicación o programa no libera correctamente la memoria que ha asignado previamente en el heap (montón). Esto puede suceder por errores de programación, como no desasignar memoria, o por referencias circulares. Las consecuencias incluyen un consumo excesivo de memoria que puede bloquear el sistema operativo, ralentizar el funcionamiento de la aplicación y generar errores inesperados.

2. ¿Qué es el Patrón RAII?, que resuelve?

RAII son las siglas de Resource Acquisition Is Initialization (La Adquisición de Recursos es Inicialización). Es un modismo de C++ que vincula el ciclo de vida de un recurso (como memoria dinámica, archivos o sockets) al ciclo de vida de un objeto.

El recurso se adquiere en el constructor del objeto y se libera automáticamente en su destructor.

Cuando hablamos de resolver, se refiere a garantizar que los recursos se liberen automáticamente cuando el objeto sale de su ámbito (scope), evitando fugas de recursos y asegurando la seguridad ante excepciones.

3. ¿Qué es el Principio de Exclusión Mutua?

Este principio se aborda a través de la gestión de bloqueos (mutex) en programación multiproceso utilizando RAII. Se implementa mediante clases como **std::lock_guard,** que adquieren un mutex (bloqueo) al crearse y lo liberan automáticamente al destruirse. Esto garantiza que el recurso compartido se desbloquee siempre, incluso si ocurre una excepción, evitando bloqueos indefinidos (deadlocks).

4. ¿Qué entiende por Excepción?

Una excepción es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones. Es un mecanismo proporcionado por C++ para manejar errores en tiempo de ejecución, permitiendo separar el código de manejo de errores del código normal.

5. Explicar: Try – Catch and Throw, dentro del ciclo de vida de una excepción.

Throw: Se utiliza para "lanzar" una excepción cuando se detecta un problema o error.

Try: Es un bloque de código que envuelve las instrucciones que podrían generar (lanzar) una excepción. Se utiliza para "intentar" ejecutar el código propenso a errores.

Catch: Es el bloque que "atrapa" y maneja la excepción si esta ocurre dentro del bloque try. Aquí se define cómo responder al error.

6. Comparar RAII Vs Excepciones, cuando usar cada uno?

Aunque ambos ayudan a gestionar errores y recursos, tienen propósitos distintos:

RAII:

Se debe usar para la gestión automática de recursos (memoria, archivos, mutex).

La ventaja es determinista, predecible y evita fugas de recursos incluso si hay errores.

En tanto a la relación, RAII es fundamental para escribir código seguro ante excepciones (exception-safe).

Excepciones:

Se deben usar para notificar errores en tiempo de ejecución o situaciones inesperadas (como errores de E/S o matemáticos) que no pueden manejarse localmente.

La ventaja que posee es que separan el código de error de la lógica principal y propagan el error hacia arriba en la pila de llamadas.

7. Describir Punteros vs Referencias, realizar una Comparativa de sus usos.

Punteros: Son variables que almacenan direcciones de memoria. Pueden ser nulos (nullptr), reasignarse para apuntar a otros objetos y requieren desreferenciación (*) para acceder al valor. Se usan para gestión dinámica de memoria y estructuras de datos opcionales.

Referencias: Son alias (nombres alternativos) para una variable existente. Deben inicializarse al declararse, no pueden ser nulas ni reasignarse. Se usan sintácticamente igual que la variable original. Son ideales para pasar parámetros a funciones y valores de retorno evitando copias.

8. ¿Cómo evitar la pérdida de memoria?

Para evitar la pérdida de memoria en C++ de manera efectiva, la clave está en adoptar buenas prácticas de gestión de recursos, comenzando por el uso del patrón RAII (Resource Acquisition Is Initialization), que vincula el ciclo de vida de los recursos al de los objetos, asegurando su liberación automática. Es fundamental reemplazar la gestión manual de memoria a través de new y delete por punteros inteligentes como std::unique_ptr y std::shared_ptr, los cuales eliminan la necesidad de liberar memoria explícitamente y previenen errores comunes. Además, se recomienda utilizar contenedores estándar como std::vector o std::string, ya que gestionan su propia memoria internamente. También es vital prestar atención a las referencias circulares, especialmente al usar punteros compartidos, problema que se puede mitigar utilizando std::weak_ptr. Finalmente, el uso de herramientas de diagnóstico como Valgrind es esencial para detectar y corregir cualquier fuga que pueda haber pasado desapercibida durante el desarrollo.

9. ¿Qué son los punteros inteligentes y qué problema soluciona?

Los Smart pointers son objetos que se comportan como punteros tradicionales pero gestionan automáticamente el tiempo de vida de los objetos a los que apuntan.

Pueden solucionar el problema que eliminan la necesidad de liberar memoria manualmente (el delete explícito), previniendo fugas de memoria (memory leaks)

y problemas de punteros colgantes (dangling pointers) al liberar la memoria automáticamente cuando el puntero sale de ámbito.

10. Describir los 3 Tipos de Punteros Inteligentes.

unique_ptr: Representa una propiedad exclusiva. No se puede copiar, solo mover (move semantics). Cuando el puntero se destruye, se elimina el objeto apuntado.

shared_ptr: Representa una propiedad compartida. Mantiene un contador de referencias; el objeto apuntado solo se elimina cuando el último shared_ptr que lo apunta es destruido (el contador llega a cero).

weak_ptr: Es una referencia débil a un objeto gestionado por un shared_ptr. No aumenta el contador de referencias y no posee el objeto. Se usa para observar el objeto sin evitar su destrucción.

11. Realizar una Comparativa de Ellos y en qué casos los Utilizaría.

unique_ptr	shared_ptr	weak_ptr
Cuando necesitas un único propietario del recurso y máximo rendimiento (poca sobrecarga). Es el puntero por defecto recomendado.	Cuando varios objetos necesitan compartir la propiedad de un recurso y no se sabe cuál terminará último (por ejemplo, en estructuras de datos complejas o grafos). Tiene sobrecarga por el conteo de referencias.	Para romper referencias circulares que causarían fugas de memoria con shared_ptr, o para implementar cachés donde no quieres impedir que el objeto se elimine si nadie más lo usa.