Informe Final Desafío II

Duvan Camilo Aragón Gallego

TI. 1033183697

David Fernando Revelo Morales

C.C 1085909373

Informática II

Aníbal Jose Guerra Soler

Universidad de Antioquia

Medellín

2025

1. Introducción

Este documento presenta el informe final del Desafío II, en el que desarrollamos una aplicación en C++ llamada UdeATunes. El objetivo del proyecto era simular cómo funciona una plataforma de streaming musical, gestionando usuarios, artistas, álbumes, canciones y anuncios. Todo esto lo hicimos aplicando programación orientada a objetos y controlando la memoria de forma manual.

Durante el desarrollo, implementamos varios conceptos importantes de la materia: encapsulamiento, herencia, composición y manejo de punteros y arreglos dinámicos, todo sin usar librerías como STL. El sistema además incluye un módulo propio para medir el consumo de recursos, registrando el número de iteraciones ejecutadas y la memoria dinámica utilizada durante la ejecución.

En este informe explicamos en detalle todo el proceso, desde que analizamos el problema hasta que hicimos las pruebas finales del sistema. También se documenta el diseño de clases, la estructura de datos empleada, los resultados obtenidos en la medición de rendimiento y las conclusiones generales del trabajo.

2. Objetivos

Objetivo general:

Crear una aplicación en C++ que simule una plataforma de música con funciones para gestionar usuarios, canciones, artistas y anuncios, aplicando los principios de POO y midiendo el uso de recursos del sistema.

Objetivos específicos:

- Entender qué necesitaba el sistema UdeATunes y definir las entidades principales y cómo se relacionan entre sí.
- Diseñar una estructura modular y orientada a objetos que garantice organización y escalabilidad.
- Programar la solución usando memoria dinámica y punteros, sin recurrir a las estructuras del STL.
- Incorporar un componente de medición que contabilice iteraciones y memoria empleada.
- Evaluar el desempeño del sistema mediante pruebas funcionales y de carga, analizando la eficiencia del código.

3. Análisis del problema

El mayor reto de este desafío fue crear una aplicación que simulara una plataforma de streaming, pero con varias restricciones técnicas:

- No se podían utilizar estructuras de datos del STL (como vector, map, etc.).
- Toda la gestión de colecciones debía realizarse mediante arreglos dinámicos.
- El programa debía incorporar un mecanismo de medición de rendimiento (iteraciones y memoria).

Con estas limitaciones, el sistema tenía que poder:

- Registrar y manejar usuarios (estándar y premium).
- Administrar artistas, álbumes y canciones.
- Reproducir canciones de manera aleatoria, incluyendo anuncios en el caso de usuarios estándar.
- Guardar y recuperar datos a partir de múltiples archivos de texto estructurados con delimitadores personalizados (principalmente punto y coma(;), y pipes (|)). Esta arquitectura de persistencia utiliza archivos separados por entidad (*usuarios.txt, artistas.txt, albums.txt, canciones.txt, creditos.txt, anuncios.txt, favoritos.txt*) en lugar de un archivo único, por las siguientes razones:
    - **Normalización de datos**: Cada dato se almacena una sola vez, evitando redundancia masiva. Por ejemplo, la información de un artista no se repite en cada una de sus canciones.
    - **Eficiencia en operaciones**: Permite carga selectiva de datos. Para validar un inicio de sesión, sólo se lee *usuarios.txt* en lugar de cargar todo el sistema.
    - **Separación de responsabilidades**: Modificaciones en la estructura de una entidad (ej: usuarios) no afectan otras entidades (ej: canciones).
    - **Integridad referencial**: Los archivos se relacionan mediante identificadores únicos (similar a llaves foráneas en bases de datos), manteniendo consistencia entre entidades.
    - **Escalabilidad y mantenimiento**: Facilita agregar nuevas funcionalidades o entidades sin reescribir todo el sistema de persistencia.
- Mostrar estadísticas de uso y de reproducción.

Estrategia de Búsqueda y Estructuras de Datos para Eficiencia

Dado que el desafío prohíbe el uso de STL y requiere medición de eficiencia (iteraciones y memoria), fue crucial analizar diferentes métodos de búsqueda y seleccionar las estructuras de datos óptimas para cada caso de uso.

Métodos de Búsqueda Considerados

Se evaluaron tres enfoques principales:

Método	Complejidad Temporal	Requisitos	Ventajas	Desventajas
Búsqueda Lineal	O(n)	Ninguno	Simple, funciona con datos desordenados	Lenta para grandes volúmenes
Búsqueda Binaria	O(log n)	Datos ordenados	Rápida, sin overhead de memoria	Requiere ordenamiento previo
Arreglos Asociativos	O(n) peor caso	Implementación propia compleja	Flexible para claves no numéricas	Sin STL, requiere búsqueda lineal

Decisiones de Implementación por Caso de Uso

a. Búsqueda de Usuarios (por nickname) - Inicio de sesión

Método elegido: Hash Table propia con encadenamiento

Justificación: El inicio de sesión es una operación crítica. Con potencialmente miles de usuarios, una búsqueda lineal requeriría hasta 10,000 iteraciones en el peor caso. Manteniendo el arreglo ordenado alfabéticamente, la búsqueda binaria reduce esto a ~14 iteraciones ($\log_2$ 10,000). El costo de ordenamiento se paga una sola vez al cargar los datos desde archivo. Aunque no es O(1) como una hash table del STL, O(log n) es suficientemente eficiente para esta operación.

b. Búsqueda de Canciones (por ID de 9 dígitos)

Método elegido: Búsqueda Binaria en arreglo ordenado por ID

Justificación: Los IDs de canciones siguen la estructura AAAAA-BB-CC (artista-álbum-canción) y se generan secuencialmente al cargar datos, por lo que el arreglo queda naturalmente ordenado. Con 100,000+ canciones, búsqueda lineal requeriría 50,000 iteraciones promedio, mientras que búsqueda binaria solo requiere ~17 iteraciones ($\log_2$ 100,000). Aunque una hash table sería O(1), requeriría ~800KB adicionales de memoria. La búsqueda binaria ofrece excelente rendimiento (5,882x más rápida que lineal) sin costo adicional de memoria.

c. Búsqueda de Canciones de un Álbum específico

Método elegido: Búsqueda de rango optimizada

Justificación: Aprovechando la estructura jerárquica de los IDs (AAAAA-BB-CC), podemos calcular el rango de IDs válidos para un álbum (ej: 123450201 a 123450299 para álbum 02 del artista 12345). Usamos búsqueda binaria para encontrar el primer ID del rango O(log n), luego recorrido secuencial para obtener todas las canciones del álbum O(m), donde m es típicamente 10-15 canciones. Complejidad total: O(log n + m), significativamente mejor que O(n) de búsqueda lineal.

d. Búsqueda de Artistas

Método elegido: Estrategia dual (Búsqueda binaria por ID + Hash table por nombre)

Justificación: Existen dos casos de uso distintos: (1) búsqueda por ID al cargar álbumes relacionados, y (2) búsqueda por nombre desde la interfaz de usuario. Mantener el arreglo principal ordenado por ID permite búsqueda binaria O(log n) para el primer caso. Adicionalmente, una hash table auxiliar con nombres como clave permite búsqueda O(1) para el segundo caso. El costo de ~50KB adicionales para 5,000 artistas es mínimo comparado con la flexibilidad y velocidad obtenida.

e. Lista de Favoritos (verificar duplicados)

Método elegido: Arreglo ordenado con búsqueda binaria e inserción ordenada

Justificación: Los usuarios premium pueden tener hasta 10,000 canciones favoritas. Verificar duplicados con búsqueda lineal requeriría 5,000 iteraciones promedio por cada operación "agregar". Manteniendo el arreglo ordenado por ID de canción, la búsqueda binaria reduce esto a ~14 iteraciones máximo. Aunque la inserción ordenada tiene costo O(n) en peor caso, esta operación es menos frecuente que las búsquedas, y el balance es favorable. Una hash table consumiría ~80KB adicionales por usuario premium, lo cual es excesivo.

La selección estratégica de métodos de búsqueda según el caso de uso específico permite cumplir con los requisitos de eficiencia del desafío, balanceando complejidad temporal, consumo de memoria y complejidad de implementación. En todos los casos, se priorizó la eficiencia en operaciones frecuentes y críticas para la experiencia del usuario.

Cada tipo de usuario debía comportarse de forma distinta: los usuarios premium disfrutaban de la reproducción continua sin interrupciones, mientras que los usuarios estándar debían escuchar anuncios entre canciones.

Básicamente, queríamos hacer una simulación completa con código bien organizado y modular, respetando todas las restricciones del desafío.

4. Diseño del Sistema

Diseñamos UdeATunes de forma modular: cada clase representa una parte específica del sistema. El modelo final está compuesto por los siguientes elementos principales:

Clase	Descripción general	Relación principal
Usuario	Representa a los usuarios del sistema de streaming. Almacena datos personales, credenciales de acceso, tipo de membresía (estándar o premium), ubicación geográfica y fecha de registro. Gestiona las listas de reproducción personalizadas y el historial de reproducción del usuario.	Se relaciona con *Plataforma* mediante agregación y con *SistemaReproduccion* mediante asociación para iniciar sesiones de reproducción.
Artista	Define un artista musical con su información biográfica, demográfica y métricas de popularidad. Mantiene un catálogo completo de todos los	Contiene uno o varios *Album* mediante composición (relación 1 a muchos). Un artista puede tener múltiples álbumes, pero cada álbum pertenece a un único artista.

	álbumes publicados por el artista en la plataforma.	
Album	Agrupa canciones de un mismo artista en una colección musical cohesiva. Almacena metadatos del álbum incluyendo géneros musicales (hasta 4), información del sello disquero, duración total, portada y calificación de usuarios.	Contiene varias *Cancion* mediante composición (relación 1 a muchos). Pertenece a un único *Artista.*
Cancion	Representa una pista musical individual con sus atributos únicos como título, duración, ubicación de archivos de audio en diferentes calidades (128 kbps y 320 kbps) y contador de reproducciones. El identificador de 9 dígitos codifica la jerarquía: artista-álbum-canción.	Asociada a *Album* y *Credito* mediante composición. Utilizada por *SistemaReproduccion* durante la reproducción.
Credito	Indica la participación de personas en la creación de una canción, diferenciando roles específicos (compositor, productor, músico, etc.). Almacena información de identificación y afiliación legal para el cálculo de regalías según derechos de autor.	Asociado a *Cancion* mediante composición (relación muchos a muchos: una canción puede tener múltiples créditos y una persona puede tener créditos en múltiples canciones).
Anuncio	Define un mensaje publicitario con su contenido textual (máximo 500 caracteres), nivel de prioridad basado en categorías (C, B, AAA) y ponderación asociada. Los anuncios se muestran intercalados durante la	Utilizado durante la reproducción por *SistemaReproduccion.* La plataforma puede mantener hasta 50 anuncios activos simultáneamente.

	reproducción para usuarios estándar.	
Plataforma	Clase principal que gestiona todas las colecciones del sistema: usuarios registrados, artistas disponibles, álbumes publicados y anuncios activos. Actúa como punto de entrada único para operaciones de alto nivel como búsquedas, autenticación y gestión de datos.	Contiene referencias a la mayoría de las entidades del sistema mediante agregación. Coordina la interacción entre *Usuario* y *SistemaReproduccion.*
SistemaReproduccion	Controla toda la lógica de reproducción de canciones, gestión de anuncios publicitarios y simulación del comportamiento de la aplicación. Maneja estados de reproducción (play/pause/stop), cola de reproducción, historial, modo aleatorio y modo repetir. Aplica las reglas de negocio diferenciadas por tipo de usuario.	Interactúa con *Usuario*, *Cancion* y *Anuncio*. Aplica lógica diferenciada según el tipo de membresía del usuario (calidad de audio, anuncios, funcionalidades premium).

Aplicamos encapsulamiento y composición, haciendo que la clase *Plataforma* sea el centro del sistema y que distribuya tareas a las otras clases.

5. Estructuras de datos empleadas

Uno de los retos más importantes fue no poder usar las estructuras de colecciones del STL (como *vector, map, list,* etc.). Tuvimos que implementar toda la gestión de colecciones manualmente usando arreglos dinámicos, punteros y manejo explícito de memoria. La única excepción permitida fue el uso de *std::string* para el manejo de cadenas de texto.

5.1 Arreglos dinámicos con redimensionamiento automático

Para todas las colecciones implementamos la estrategia de arreglos dinámicos con capacidad variable. Cada clase que maneja colecciones mantiene tres elementos clave:

- Un puntero al arreglo de elementos (por ejemplo: *Usuario** usuarios*)
- La cantidad actual de elementos almacenados (*cantidadUsuarios*)
- La capacidad máxima actual del arreglo (*capacidadUsuarios*)

Cuando se intenta agregar un elemento y el arreglo está lleno, se duplica automáticamente la capacidad: se crea un nuevo arreglo más grande, se copian todos los elementos existentes, se libera el arreglo antiguo y se actualiza el puntero. Esta estrategia garantiza un crecimiento eficiente con complejidad amortizada O(1) para inserciones.

El patrón se aplicó de forma consistente en todas las clases que manejan colecciones:

- Plataforma: gestiona arreglos de usuarios, artistas y anuncios
- Artista: gestiona arreglo de álbumes
- Album: gestiona arreglo de canciones
- Cancion: gestiona arreglo de créditos
- ListaFavoritos: gestiona arreglo de punteros a canciones

5.2 Estrategias de indexación para búsquedas eficientes

Dado que no podíamos usar estructuras como *map* o *unordered_map*, implementamos índices auxiliares para optimizar las búsquedas frecuentes:

Índice por clave primaria: Para entidades con identificadores únicos (como usuarios con nickname, canciones con ID), mantuvimos arreglos auxiliares de punteros ordenados por dicho campo. Esto nos permite aplicar **búsqueda binaria** en lugar de búsqueda lineal, reduciendo la complejidad de O(n) a O(log n).

Por ejemplo, en la clase Plataforma:

- Un índice *usuariosOrdenadosPorNickname* permite encontrar usuarios rápidamente por su nombre
- Un índice *cancionesOrdenadasPorID* facilita la búsqueda de canciones por identificador

Lazy sorting: Los índices no se reordenan después de cada inserción individual. En cambio, mantenemos una bandera booleana que indica si el índice está desactualizado. Solo cuando se va a realizar una búsqueda y la bandera indica que hay cambios pendientes, se ejecuta el ordenamiento. Esto optimiza el caso común donde se insertan múltiples elementos seguidos antes de realizar búsquedas.

Algoritmo de ordenamiento: Implementamos el algoritmo QuickSort para ordenar los índices, con complejidad promedio O(n log n). La función de comparación varía según el tipo de índice (por nickname, por ID numérico, etc.).

5.3 Uso estratégico de std::string

Aunque las estructuras de colecciones estaban prohibidas, se permitió el uso de *std::string* para el manejo de cadenas de texto. Esto simplificó significativamente:

- La gestión de memoria de textos (no hay que usar *new[]/delete[]* manualmente)
- Las operaciones de comparación y concatenación de cadenas
- La prevención de errores comunes como buffer overflow
- La compatibilidad con funciones de entrada/salida

Todos los atributos de tipo texto en nuestras clases (nombres, rutas, descripciones, etc.) se declararon como *std::string*, lo que redujo la complejidad del código y mejoró la robustez del sistema.

5.4 Completitud del sistema de indexación

El sistema de indexación está completo para todas las búsquedas críticas que requieren las funcionalidades del sistema:

✓ **Usuarios**: Indexados por nickname para autenticación rápida y búsqueda O(log n)
✓ **Canciones**: Indexadas por ID de 9 dígitos para acceso directo en reproducción
✓ **Artistas**: Indexados por código identificador de 5 dígitos para consultas de catálogo
✓ **Álbumes**: Accesibles eficientemente a través del arreglo de su artista propietario

Índices no implementados: No se crearon índices secundarios para búsquedas menos frecuentes como filtrado por género musical, año de lanzamiento o puntuación de álbum, ya que no eran requeridos por las funcionalidades principales del desafío. Para estas búsquedas ocasionales, se utiliza recorrido lineal, lo cual es aceptable dado que no son operaciones críticas del sistema.

La arquitectura modular diseñada permite agregar nuevos índices fácilmente en el futuro sin modificar las clases base, simplemente extendiendo la clase *Plataforma* con nuevos arreglos auxiliares.

5.5 Gestión de memoria y prevención de fugas

Sin el uso de contenedores STL, la gestión manual de memoria requirió especial atención:

- **Destructores robustos**: Cada clase con memoria dinámica implementa su destructor que libera todos los arreglos con *delete[]* y todos los objetos individuales con *delete*.
- **Regla de los tres**: Para clases con recursos dinámicos implementamos constructor de copia, operador de asignación y destructor, siguiendo la "regla de los tres" de C++.
- **Ownership claro**: Definimos claramente qué clase es "dueña" de cada objeto. Por ejemplo, Plataforma es dueña de todos los usuarios y artistas, mientras que *ListaFavoritos* solo guarda punteros pero no es dueña de las canciones.
- **Liberación ordenada**: Al finalizar el programa, se destruye primero la instancia de *Plataforma*, que en cascada libera todos los usuarios, artistas, álbumes, canciones, etc.

Durante el desarrollo usamos técnicas de debugging para detectar y corregir memory leaks, double-free y accesos a memoria no válida.

5.6 Ventajas del enfoque híbrido

El uso de *std::string* combinado con estructuras manuales nos dio lo mejor de ambos mundos:

- **Seguridad**: Menos errores de manejo de memoria en strings
- **Aprendizaje**: Comprensión profunda de cómo funcionan las estructuras de datos
- **Control**: Gestión exacta de memoria en las colecciones
- **Eficiencia**: Optimizaciones específicas para nuestros patrones de acceso

Este enfoque híbrido demostró que es posible construir sistemas complejos sin depender completamente del STL, mientras se aprovechan sus componentes más seguros y probados.

6. Funcionalidades implementadas

Logramos implementar todas las funcionalidades principales que pedía el desafío. A continuación se resumen las más destacadas:

- **Carga de datos automática:**
  Al iniciar el programa, la clase *Plataforma* lee los archivos CSV y carga todos los registros en memoria, estableciendo las relaciones entre usuarios, artistas, álbumes, canciones y anuncios.

- **Gestión de usuarios:**
  El sistema permite registrar nuevos usuarios, editar información y diferenciarlos según su tipo (estándar o premium).
- **Reproducción de canciones:**
  Se implementó una reproducción aleatoria de canciones, en la que los usuarios estándar escuchan anuncios cada cierto número de pistas, mientras que los usuarios premium disfrutan de una experiencia sin interrupciones.
- **Favoritos y listas de reproducción:**
  Los usuarios premium pueden marcar canciones como favoritas, y estas se guardan de forma persistente para futuras sesiones.
- **Contador de reproducciones:**
  Cada vez que se reproduce una canción, se incrementa su número de reproducciones. Esto permite obtener estadísticas de las más escuchadas.

6.1 Limitaciones identificadas

Durante el desarrollo del proyecto, identificamos algunas funcionalidades del desafío que no se pudieron implementar completamente:

Medición de recursos: El desafío solicitaba implementar un sistema de medición que contabilizara:

- El número de iteraciones realizadas en cada funcionalidad
- El consumo total de memoria del sistema en cada momento

Esta funcionalidad presentó desafíos significativos en su implementación:

- Contador de iteraciones: Aunque se intentó incorporar contadores en los bucles principales, mantener una contabilidad precisa y consistente en todas las funciones del sistema resultó complejo, especialmente en operaciones recursivas y llamadas entre métodos.
- Cálculo de memoria: Implementar el método *calcularMemoria()* en cada clase para sumar recursivamente todo el consumo de memoria (incluyendo arreglos dinámicos, strings y objetos anidados) requería un nivel de detalle que no se logró completar en el tiempo disponible.

Estas limitaciones no afectan la funcionalidad core del sistema (carga de datos, reproducción, listas de favoritos, etc.), pero representan un área de mejora para futuras versiones del proyecto.

En general, estas funciones hacen una buena simulación de una plataforma de música, cumpliendo con todos los requisitos del desafío.

7. Conclusiones

Desarrollar UdeATunes fue un verdadero reto porque tuvimos que aplicar POO sin usar las librerías modernas del lenguaje. "Esto nos ayudó a entender mucho mejor cómo funcionan los punteros, los arreglos dinámicos y el manejo de memoria, además de mejorar nuestra forma de diseñar código modular.

A nivel funcional, el programa cumple con los objetivos planteados:

- Se logró modelar correctamente las entidades principales del dominio (usuarios, canciones, artistas, álbumes y anuncios).
- Se implementó un sistema de reproducción con comportamiento diferenciado para usuarios estándar y premium.
- La lectura y escritura de datos mediante archivos CSV se ejecutó correctamente, asegurando la persistencia de la información.

Más allá de lo técnico, este proyecto nos ayudó a mejorar habilidades importantes: organizar mejor el código, analizar problemas reales y convertir requisitos en soluciones que funcionen.

En conclusión, el proyecto cumple con casi todo lo que pedía el Desafío II, demostrando que entendimos y aplicamos bien los conceptos de la materia.

8. Reflexiones personales sobre el desafío

Este proyecto nos sirvió para aplicar lo que aprendimos en clase, pero también nos dimos cuenta de lo complicado que es manejar los recursos manualmente. Al no poder usar estructuras del STL, fue necesario planear cuidadosamente la forma en que los datos serían almacenados, accedidos y liberados, lo que ayudó a fortalecer el pensamiento lógico y la disciplina al programar.

Hacer la medición de recursos nos hizo valorar más el rendimiento y la eficiencia, cosas que uno a veces ignora en proyectos más simples. Durante las pruebas fue interesante observar cómo pequeñas decisiones en el código podían afectar notablemente la cantidad de iteraciones o el consumo de memoria.

Además, trabajar en equipo fue un componente clave: coordinar las partes del código, distribuir tareas y combinar estilos de programación distintos ayudó a mejorar la comunicación y la planificación. En definitiva, UdeATunes no solo fue un reto técnico, sino también una experiencia formativa que contribuyó al crecimiento profesional y personal de quienes participaron en su desarrollo.