1. Análisis general del proyecto

El trabajo consistió en desarrollar un sistema llamado UdeATunes, una simulación de una plataforma de streaming musical, diseñada completamente en C++.
La idea principal fue crear una estructura que representara de forma lógica cómo se relacionan los diferentes elementos de un servicio de música: usuarios, artistas, álbumes, canciones, productores, créditos y publicidad.

Además de eso, el programa debía diferenciar entre usuarios estándar y VIP, donde los VIP tienen más libertades (como escuchar sin anuncios o acceder a listas personalizadas), mientras que los usuarios estándar reciben publicidad entre canciones.

Todo el proyecto se desarrolló sin usar estructuras de datos de la STL, ya que ese era un requisito del desafío. Por eso se implementó una clase propia llamada MiLista<T>, que maneja los elementos dinámicamente con punteros, garantizando que la memoria se gestione de forma controlada.

En general, UdeATunes busca demostrar el uso de programación orientada a objetos, plantillas, modularidad y manejo eficiente de memoria dinámica.

2. Diseño de las clases y relaciones
La arquitectura del sistema se construyó a partir de las siguientes clases principales:
UdeATunes: clase principal que contiene todos los usuarios y artistas.
Usuario: representa a cada persona dentro del sistema; puede ser estándar o VIP (controlado por un bool membresía).
Artista: contiene los álbumes que ha creado.
Álbum: contiene las canciones que pertenecen a ese álbum y calcula su duración total.
Canción: tiene su duración, rutas de audio (128 kbps y 320 kbps), y los créditos que la componen.
Créditos: registra nombres, apellidos, tipo de participación (productor, disquera, patrocinador) y código de afiliación.
Publicidad: administra los anuncios con ponderaciones para determinar su aparición.
MiLista<T>: reemplaza estructuras como vector o list, siendo un contenedor genérico de punteros dinámicos.
Cada clase fue diseñada para reflejar una relación de uno a muchos, es decir: UdeATunes → Usuarios y Artistas → Álbumes → Canciones → Créditos.

De esta manera, todo el sistema se organiza jerárquicamente desde la clase principal hasta los elementos individuales.

3. Desarrollo y decisiones técnicas

Desde el inicio del desarrollo fue necesario estructurar las clases con constructores claros y funciones bien definidas, ya que los errores más comunes venían de dependencias cruzadas entre archivos.h.
Por eso se cuidó el orden de los #include y el uso de plantillas para que la compilación fuera estable.
El sistema de publicidad fue uno de los apartados más interesantes, ya que se implementó con un algoritmo ponderado que elige los anuncios de forma aleatoria pero con prioridad según su tipo.
Esto significa que un anuncio “A” tiene más probabilidad de mostrarse que uno tipo “C”.
También se controló que un anuncio no se repita inmediatamente.
En el caso de las canciones, se manejan dos rutas de reproducción: una en 128 kbps y otra en 320 kbps.
Los usuarios VIP siempre reproducen la versión de mayor calidad.
Además, se incluyeron funciones como pausar, pasarCancion, regresarCancion, aleatorio y misFavoritos.
El sistema MiLista maneja los datos internamente usando punteros dobles (T**) y redimensionamiento automático, lo que permite manejar centenares o miles de elementos sin problemas.
Esto fue esencial porque se especificó que el sistema debía soportar listas grandes, incluso de más de 10.000 canciones.

4. Dificultades encontradas y soluciones

Durante el desarrollo aparecieron varios errores que ayudaron a entender mejor el funcionamiento del lenguaje y la estructura de memoria.
Uno de los más recurrentes fue el de constructores no declarados, especialmente al compilar las clases Artista, Álbum y Créditos.
También surgieron errores por dependencias circulares, que se corrigieron organizando correctamente los #include y aplicando el principio de modularidad.
Otro error importante fue en el manejo del hilo Kron, el cual servía para guardar los datos en segundo plano.

En un principio el programa entraba en un bucle infinito porque el hilo se había creado con detach(), lo que hacía que siguiera corriendo incluso después de que el objeto principal se destruyera.
La solución fue controlar el ciclo con una variable booleana y reemplazar detach() por join(), de modo que el hilo pudiera cerrarse de forma segura.
También se detectó un error en el algoritmo de publicidad: return anuncios[n + 1];
Esto generaba un acceso fuera de rango.
Se corrigió cambiándolo por: return anuncios[n - 1];

5. Resultado final del sistema

El sistema UdeATunes funciona correctamente y cumple con lo pedido en el desafío:

Maneja usuarios, artistas, álbumes y canciones.
Distingue entre usuarios estándar y VIP.
Controla la publicidad aleatoria según prioridad.
Permite la reproducción en distintas calidades.
Implementa un contenedor genérico propio (MiLista<T>).
Mide memoria usada y controla los recursos dinámicos.

Todo el código se encuentra versionado en GitHub, con commits progresivos que reflejan cada fase del desarrollo, desde la creación de las clases hasta la depuración final.

6. Conclusión personal

Este proyecto me permitió comprender de manera más clara y profunda la lógica detrás de los sistemas de cómo cada clase debe encajar como un elemento de un todo.
Aprendí la importancia del orden, la claridad del código y la gestión responsable de memoria.
También comprendí que escribir código no es solo hacerlo funcionar, sino hacerlo limpio, entendible y eficiente.

UdeATunes no solo me ayudó a reforzar la parte técnica, sino también la forma de pensar en cómo los programas se comunican internamente y cómo cada decisión en el diseño afecta el rendimiento y la estabilidad.