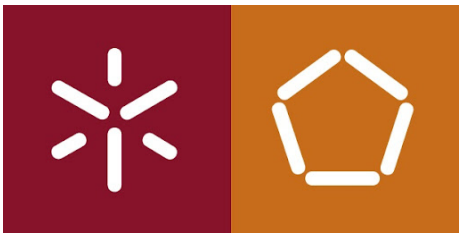

**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

# Laboratórios de Informática III

## Trabalho prático (Fase 1)

**Outubro 2024**

Desenvolvido por:

Rodrigo Ferreira - a104531
Gonçalo Alves - a104079
Diogo Esteves - a104004

# Contents

# 1 Introdução

Este relatório tem como finalidade detalhar a estrutura e as metas do projeto prático de LI3, que envolve a criação de um sistema de *streaming* de música. Este projeto tem como objetivo aprimorar habilidades fundamentais de C, concentrando-se na modularização e no encapsulamento do código e ainda o uso do gdb e valgrind.

O projeto desenvolvido é fundamentado num conjunto de dados que inclui informações sobre as entidades que achámos convenientes, ou seja, músicas, artistas e utilizadores. O objetivo é armazenar esses dados em estruturas apropriadas, o que possibilita a realização de consultas específicas para extrair e manipular as informações conforme solicitado. A etapa inicial deste projeto centra-se no *parsing* e validação de um dataset contendo utilizadores, artistas, músicas e, por fim, sobre o qual serão executadas queries.

A modularização do código em componentes de interface e implementação, a validação e *parsing* de dados, além do processamento eficaz de consultas, são as características mais notáveis do sistema. Entre as principais características do sistema, destacam-se a separação clara de responsabilidades entre os módulos, como o *Database Manager* que centraliza o acesso aos dados, permitindo uma gestão eficiente e encapsulada das informações de artistas, músicas e utilizadores. A utilização dos módulos *Artist Manager*, *Music Manager* e *User Manager* facilita a manipulação específica de cada tipo de entidade, garantindo que as operações realizadas sejam coesas e isoladas, o que contribui para a robustez do sistema. Portanto, este projeto cria um ambiente sólido para operações de manipulação e avaliação de dados num sistema de *streaming* de música, pavimentando assim o caminho para funcionalidades mais sofisticadas e uma otimização mais intensa nas etapas futuras.

# 2 Sistema

Para facilitar a compreensão da estrutura da nossa aplicação, segue o seguinte diagrama:

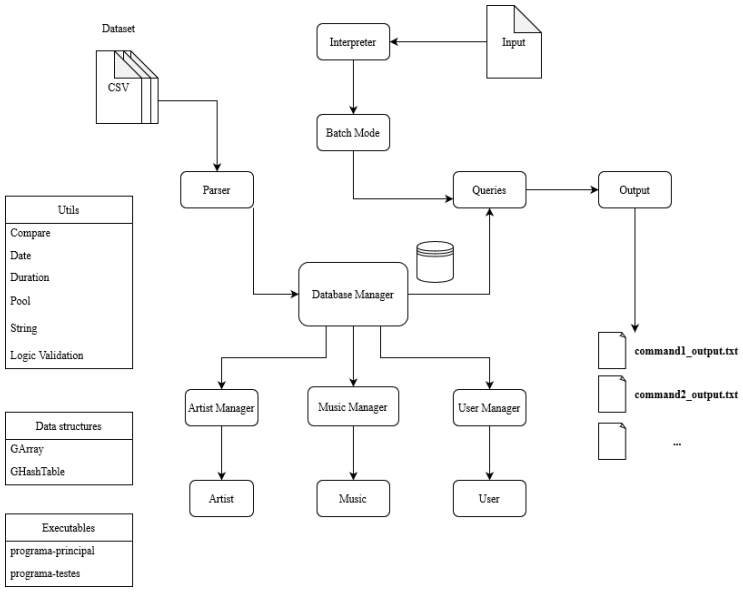


Figure 1: Arquitetura da aplicação

## 2.1 Managers

Os módulos responsáveis pela manipulação de dados em memória são os *managers* de dados. Estes estão organizados hierarquicamente de forma a encapsular os diferentes tipos de dados e assim conferir uma robustez ao sistema.

Os *managers* de mais baixo nível têm como objetivo manipular as operações de inserção e leitura nas suas respetivas *Hash Tables*. Por outro lado, o *manager* de mais alto nível (Database Manager), tem como principal objetivo conferir ao sistema uma ferramenta universal de acesso a todo o *Dataset* durante a sua execução. Este serve-se das funções disponibilizadas pelos *managers* de mais baixo nível, para executar as operações de inserção e leitura dos dados armazenados em memória.

## 2.2 Parsing

O módulo *parser* é o responsável por fazer a leitura dos ficheiros *CSV* presentes no *Dataset* e armazená-los em memória, com o auxilio do módulo *Database*

*Manager*. À medida que a leitura dos ficheiros, linha a linha, é executada, é feita a validação sintática dos diferentes campos de cada linha, com o auxílio dos *setters* associados a cada um dos tipos de dados a armazenar (*users, artists e musics*). Para além disso, à medida que vamos inserindo na estrutura temporária ao usarmos *setters* resolvemos já a parte de retirar as aspas e as barras, de forma a guardar apenas o conteúdo de cada string.

Após a leitura de cada linha, uma estrutura de dados auxiliar, com o tipo adequado ao ficheiro a ser lido, está devidamente preenchida com as informações necessárias à realização das queries propostas. O passo final é fazer a validação lógica da estrutura, antes de fazer a sua inserção em memória. Para isso, é utilizado o módulo de *logic validation* para validar os dados introduzidos e, em caso de sucesso, a mesma é clonada para a estrutura de dados adequada (como uma *GHashTable*).

## 2.3 Queries

A execução das queries propostas é garantida pelos módulos *query1, query2, query3*. Estes, através do módulo *Database Manager* têm acesso aos dados necessários para a execução da tarefa pretendida e devolvem como resultado uma *string* com o output pretendido, que será escrito no devido ficheiro de *output* pelo módulo de output.

## 2.4 Output

O módulo de *output* tem como função a escrita em ficheiros .txt da *string* pretendida. Durante o *parsing*, sempre que uma linha contém dados inválidos, é este o módulo responsável pela escrita desta no devido ficheiro de erros. No decorrer da execução daos *queries*, é este o módulo responsável pela escrita do resultado de cada uma das queries.

# 3 Discussão

## 3.1 Modularização e Encapsulamento

Nesta 1º fase do trabalho, começámos por desenhar a arquitetura sendo posteriormente mais fácil na implementação do mesmo. Os *managers* e as entidades contribuiram para a clareza e manutenção do código, evitando a dependência direta entre módulos. Isso permite que os *managers* cuidem da lógica de manipulação e acesso aos dados, enquanto as entidades representam apenas as estruturas de dados em si. Promove um encapsulamento claro, onde as entidades são simplificadas e a complexidade de operações fica isolada nos *managers*, o que facilita a expansão do sistema.

Encontrámos alguma dificuldade quando aumenta a modularidade, pois cresce a necessidade de manter as interfaces consistentes. Uma pequena mudança na interface exige a mudança em módulos dependentes. Exigiu cuidado para planear e comunicar mudanças com clareza.

## 3.2 Estruturas de dados

As estruturas de dados escolhidas para este projeto foram a *GHashTable* e o *GArray*. É vantajoso usar a biblioteca *Glib* neste projeto de grande dimensão, uma vez que consegue redimensionar automaticamente a memória para várias das estruturas. Além disso, oferece funções para libertar memória adequadamente, o que reduz riscos de *leaks* e ainda funções utilitárias como funções de iteração.

Porém poderiamos ter melhorado na abordagem das estruturas de dados, mas não demos muito foco. Focámo-nos mais no GArray, pois armazena os elementos de forma contígua na memória e como fazemos bastantes leituras é eficiente (O(1)), permitindo o acesso direto. E por outro lado, como são conjuntos grandes, a GHashTable ajuda na procura rápida com acesso por chave.

## 3.3 Execução das Queries

### 3.3.1 Query 1

Consulta de uma entidade pelo seu identificador. A sua implementação foi trivial: começa-se por determinar o tipo da entidade com base no formato do seu identificador, processo seguido da consulta direta do manager correto na base de dados.

### 3.3.2 Query 2

A query 2 tem como objetivo encontrar os top N artistas com a maior duração total de discografia, e pode ser usada com ou sem um filtro de país. Primeiramente, ela verifica se o valor de N é válido, e depois vai buscar a tabela de

artistas na base de dados. Ao iterar sobre essa tabela, a função aplica o filtro de país, caso seja necessário, e coleta os dados de cada artista, como ID, nome, tipo, duração total da discografia e país. Em seguida, ordena esses artistas com base na duração da discografia em ordem decrescente, e, em caso de empate, organiza por ID em ordem crescente. Dessa forma, a query2 fornece uma lista dos artistas com a maior discografia, indicando o tipo, a duração em horas, minutos e segundos, e o país de cada um, prontos para serem registrados no ficheiro de resultado.

### 3.3.3 Query 3

Para a query 3 tivemos bastantes otimizações feitas. A query 3 determina os géneros de música mais populares numa faixa etária específica, fornecida pelos argumentos *minAge* e *maxAge*. A função começa por validar o intervalo de idades e, em seguida, obtém os dados dos géneros musicais e as preferências dos utilizadores a partir do *User Manager*.

A query percorre os grupos de likes de utilizadores dentro da faixa etária especificada, somando o número de likes para cada género musical. Se a faixa etária incluir apenas uma idade específica, processa apenas esse grupo; caso contrário, acumula os dados de likes para todas as idades dentro do intervalo definido e ordena-os por ordem descrescente.

## 3.4 Gestão de memória

Podemos observar que alguns campos do dataset nunca precisavam de estar presentes no output de nenhuma query (ex: descrição), pelo que era escusado o seu armazenamento na base de dados. Por observação dos datasets em si, podemos concluir que é possível armazenar certos campos usando tipos de dados de menor tamanho (ex: os géneros musicais serem armazenados em enum em vez de string e inteiros de pequena dimensão em inteiros de 8 ou 16 bits).

```
typedef enum genre {
    BLUES,
    CLASSICAL,
    COUNTRY,
    ELECTRONIC,
    HIPHOP,
    JAZZ,
    METAL,
    POP,
    REGGAE,
    ROCK
} genre_t;
```

Figure 2: Enum

```
struct date{
    uint8_t day;
    uint8_t month;
    uint16_t year;
};
```

Figure 3: Data

### 3.4.1 Alocação de memória em Pools

Além de otimizações de memória no que toca aos tipos de dados escolhidos, optamos por utilizar a alocação de memória em pools para objetos do mesmo tamanho. Esta estratégia ajudou a reduzir o uso de memória em *overheads* de alocação, bem como nos permitiu controlar em parte o número de vezes que uma realocação de memória é realizada (fazendo um balanço entre o número total de objetos a armazenar e o comprimento de cada bloco de memória).

Para a 2ª fase, será também interessante desenvolver uma alocação de memória em pools para strings, de forma a otimizar ainda mais a utilização de memória do programa, tendo em conta o crescimento do *Dataset*.

## 3.5 Análise do Desempenho

Para a análise do desempenho do programa, foram utilizados diversos dispositivos, de forma a obter-mos uma visão global sobre o comportamento do mesmo em diferentes sistemas. Os dispositivos em questão foram: os computadores dos três constituintes do grupo e a plataforma de testes automáticos disponibilizada pela equipa docente. As especificações dos computadores dos constituintes do grupo estão disponíveis na secção de anexos.

### 3.5.1 Análise do tempo global e memória alocada

No que toca às execuções locais do programa, foram realizadas cinco execuções do programa, através do comando *time*, disponibilizado pelo sistema operativo e através do programa-testes, que permite uma análise individual do tempo de execução de cada etapa de execução, bem como a memória que foi globalmente utilizada pelo programa.

Assim, segue a seguinte tabela com as médias das últimas cinco execuções do programa através do comando time e da plataforma de testes:

| Dispositivo | Tempo Médio (em segundos) | Memória alocada (em MB) |
|---|---|---|
| MacBook Air M1 | 4.410 | 309.28 |
| MacBook Air M1 (Virtual Machine) | 3.604 | 453.50 |
| Acer Aspire A315-58G | 2.830 | 454.12 |
| HP Victus 16-e0019np | 3.539 | 460.25 |
| Plataforma Testes | 9.980 | 472.00 |

Table 1: Tabela de tempos médios de execução e memória alocada

O que podemos concluir acerca dos testes realizados localmente é que, dependendo do sistema utilizado (Linux vs MacOs) os tempos de execução e memória utilizada variam: no MacOs, apesar da substancial redução na memória utilizada, existe uma aumento significativo no tempo de execução do programa.

### 3.5.2 Análise através do Programa-Testes

Para obtermos uma perceção mais elaborada acerca das diferentes etapas de execução do programa, desenvolvemos um programa-testes para fazer a execução faseada do programa e assim obter os tempos de cada etapa. Segue na seguinte tabela os resultados da média das cinco execuções do programa-testes:

| Dispositivo | Artist Parsing(s) | Music Parsing(s) | User Parsing(s) | Query1(s) | Query2(s) | Query3(s) |
|---|---|---|---|---|---|---|
| MacBook Air M1 | 0.008684 | 0.281650 | 3.378385 | 0.002317 | 0.004770 | 0.002126 |
| MacBook Air M1 (Virtual Machine) | 0.005806 | 0.298711 | 2.611024 | 0.000612 | 0.002220 | 0.000452 |
| Acer Aspire A315-58G | 0.001573 | 0.164190 | 2.509430 | 0.000245 | 0.001705 | 0.000317 |
| HP Victus 16-e0019np | 0.001484 | 0.221152 | 2.879822 | 0.000477 | 0.002194 | 0.000526 |

Table 2: Tabela de tempos médios de execução do programa-testes

Graças à execução do programa-testes, retiramos duas conclusões importantes acerca da execução do programa: a primeira, é de que o a operação de *parsing* dos users é, de longe, a operação que mais pesa na execução do programa; a segunda é de que graças às estruturas de dados utilizadas nas queries, em especial na 1ª e na 3ª, é possível executar as mesmas num tempo bastante reduzido.

A última funcionalidade do programa-testes é a execução de testes unitários sobre os nossos outputs. Após a medição dos tempos de execução, este compara os nossos outputs com os outputs-esperados e, caso encontre alguma diferença, faz print no ecrã do ficheiro que difere, o que torna o processo de debug muito mais eficiente e rápido.

# 4 Conclusão

Com a realização do nosso trabalho aprofundamos na prática os conceitos de encapsulamento e modularização. A longo prazo, estes parâmetros representarão grande importância e utilidade no desenvolvimento de projetos futuros, ao nível da segurança, organização e reutilização.

Após a realização do projeto, reparámos que existem diversas melhorias a ser feitas, nomeadamente na parte do *parsing dos users*, tanto a nível de uso de memória quanto o tempo de execução. Na 2ª fase estámos cientes que vamos encontrar um desafio pela frente, uma vez que o *dataset* será exponencialmente maior irá exigir uma maior atenção na abordagem das restantes *queries* e consequentemente o menor tempo possível bem como o menor uso de memória.

# 5 Anexos

## 5.1 Especificações dos dispositivos de teste

- **MacBook Air:** Processador M1; 8GB RAM; 256GB Armazenamento
- **Acer Aspire A315-58G:** Processador Intel Core i7-1165G7; 12GB RAM; 512GB Armazenamento
- **HP Victus 16-e0019np:** Processador AMD Ryzen 5 5600H; 8GB RAM; 512GB Armazenamento