

---

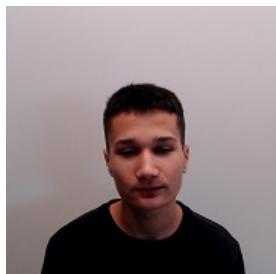
# Laboratórios de Informática III

---

TRABALHO REALIZADO POR:

BENJAMIM MELEIRO RODRIGUES  
XAVIER SANTOS MOTA  
PEDRO DANTAS DA CUNHA PEREIRA

GRUPO 46



A93323  
Benjamim Rodrigues



A88220  
Xavier Mota



A97396  
Pedro Pereira

---

# Índice

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Estruturação</b>	<b>3</b>
2.1	<i>Drivers</i>	3
2.2	<i>Users</i>	4
2.3	<i>Rides</i>	4
2.4	<i>Validator</i>	5
<b>3</b>	<b><i>Queries</i></b>	<b>7</b>
3.1	<i>Query I</i>	7
3.2	<i>Query II</i>	7
3.3	<i>Query III</i>	8
<b>4</b>	<b>Testes Funcionais</b>	<b>9</b>
<b>5</b>	<b>Encapsulamento e Modularização</b>	<b>9</b>
<b>6</b>	<b>Conclusão</b>	<b>10</b>

---

## 1 Introdução

No âmbito da unidade curricular de Laboratórios de Informática III foi-nos proposto, numa primeira fase, o desenvolvimento de uma aplicação com determinadas propriedades e funcionalidades descritas no enunciado do próprio trabalho, disponibilizado pela equipa docente da UC. Estas funcionalidades giram em torno de ficheiros de dados que contém diferentes informações relacionadas a *users*, *drivers* e *rides*, a partir das quais vamos gerar os respetivos catálogos. Para a realização deste trabalho, e como nos foi indicado inicialmente, tivemos em consideração a arquitetura proposta (Fig. 1).

Tendo em conta o propósito da UC, bem como o grande peso das componentes relacionadas à modularização e encapsulamento de dados, estes foram aspectos aos quais demos foco ao longo da realização de todo o trabalho, respeitando-os sempre e encontrando soluções que estivessem em sintonia com estes conceitos.

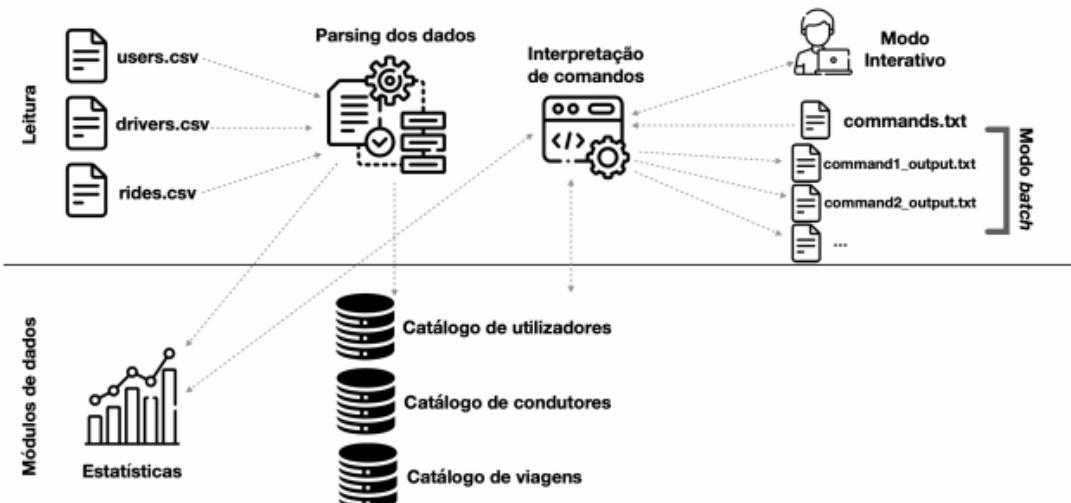


Fig. 1 - Arquitetura de referência para a aplicação a desenvolver

---

## 2 Estruturação

Em relação às estruturas utilizadas para os catálogos, tomámos diferentes decisões baseadas naquilo que consideramos ser vantajoso para o desenvolvimento do trabalho, nomeadamente para as *Queries* que definimos nesta fase do projeto.

```
struct catalog_users{
    GHashTable* users_hash;
};

struct catalog_rides{
    struct ride* array;
    size_t used;
    size_t size;
};

struct catalog_drivers{
    GHashTable* drivers_hash;
};
```

Fig. 2 - Estruturas do Catálogo de *Users*, *Rides* e *Drivers*

### 2.1 Drivers

Para o catálogo dos *Drivers*, decidímos utilizar *hashTables*, tendo em conta que, como o nosso objetivo é procurar um determinado *driver* a partir do seu *id*, este é método mais eficiente que encontrámos. Para nos auxiliar, utilizámos a biblioteca *glib*.

Para além disto, e como se pode verificar na Fig. 2, adicionámos também novos parâmetros à estrutura dos *Drivers*. Estes serão atualizados à medida que o catálogo de *Rides* é criado, de modo a facilitar a posterior execução das *queries*.

```
struct driver{

    char* id; //size>0
    char* name; //size>0
    struct tm birth_date; //dia entre 1 e 31, mês entre 1 e 12
    char* gender; //size>0
    enum car_class class; //basic, green ou premium
    char* license_plate; //size>0
    char* city; //size>0
    struct tm account_creation; //dia entre 1 e 31, mês entre 1 e 12
    enum account_status account_status; //active ou inactive

    double total_rating;
    int total_rides;
    double total_received;
    struct tm most_recent_drive;
};
```

Fig. 3 - Estrutura *Drivers*

---

## 2.2 *Users*

Para o cátalogo dos *Users* decidímos, novamente, utilizar *hashTables* com o auxílio da biblioteca *glib*. Isto porque, de forma semelhante ao que foi dito na secção dos *Drivers*, pretendemos encontrar os diferentes *Users* através do seu *username*, pelo que a utilização destas *hashTables* nos pareceu ser a maneira mais vantajosa de o fazer.

Da mesma forma que fizemos com a estrutura dos *Drivers* decidimos também adicionar variáveis à estrutura dos *Users* que também serão atualizados à medida que o catálogo de *Rides* é criado.

```
struct user{

    char* username;
    char* name;
    char* gender;
    struct tm birth_date;
    struct tm account_creation;
    char* pay_method;
    enum account_status account_status;

    double total_rating;
    int total_rides;
    double total_spent;
    int distance_total;
    struct tm most_recent_drive;
};
```

Fig. 4 - Estrutura *Users*

## 2.3 *Rides*

No caso das *Rides*, e ao contrário do que foi feito para as duas estruturas anteriores, decidímos utilizar arrays ao invés de *hashTables*. Isto porque, como no caso das *Rides* não precisamos de procurar uma *ride* através de um atributo específico (como *id*, por exemplo), não nos pareceu vantajosa a utilização de *hashTables*.

Como foi dito anteriormente, o catálogo de *Users* e *Drivers* é atualizado à medida que o catálogo das *Rides* é criado. Isto faz-se sempre que é adicionada uma *Ride* válida à estrutura de *Rides*. Para isto, criámos funções que encontram na *hashtable* de *Drivers* e na *hashtable* de *Users* o *Driver* e o *User* correspondente, respetivamente, e atualizam as variáveis destes com as informações dadas pela *Ride*.

---

```

struct ride{
    char* id; //>0
    struct tm date;
    char* driver_id; //>0
    char* user_username; //>0
    char* city; //>0
    int distance; //int >0
    double score_user; // double >= 0
    double score_driver; // double >= 0
    double tip;
    char* comment;
};

```

Fig 5 - Estrutura *Rides*

## 2.4 Validator

Para criar os elementos de cada catálogo tínhamos de percorrer três diferentes ficheiros de entrada, cada um com informação para criação dos diferentes tipos de dados constituintes de cada catálogo. No entanto, nem todos os dados nestes ficheiros eram entradas válidas de acordo com um conjunto de validações a executar.

Assim, para validar os *Users*, *Drivers* e *Rides*, de forma a certificar-mo-nos de que não guardamos dados errados em nenhum dos catálogos, criámos funções, guardando estas no ficheiro *validator.c* de forma a "reutilizar" código entre cátalogo, sendo que estas funções funcionam para todos os tipos de catálogo que desenvolvemos. Estas funções são então chamadas nos *users.c*, *drivers.c* e *rides.c* aquando da criação dos dados e caso estes não sejam considerados válidos são descartados sem ser adicionados ao catálogo.

Aproveitamos também este ficheiro para criar outras funções partilhadas por vários catálogos de forma a promover ainda mais a "reutilização" de código e a modularidade do nosso programa.

```

int verify_Size (char* token);
struct tm verify_Date(char* token);
enum account_status verify_AccountStatus (char* token);
enum car_class verify_CarClass (char* token);
int verify_Distance (char* token);
double verify_ScoreAndTip (char* token);

int calculate_Age(struct tm birth_date);
struct tm get_MostRecentDate(struct tm first_date, struct tm new_date);
int compare_DateRide(struct tm new_date, struct tm first_date);
double calculate_RatingAverage(double total_rating, int total_rides);

```

Fig 6 - Funções da *Validator*

---

Para as datas usadas ao longo das diferentes estruturas decidimos usar a *struct tm* definida na *Library <time.h>*.

Quanto ao **Account\_Status** & à **Car\_Class**, criamos um *Enum* como podemos ver na figura 7.

```
enum account_status{
    NoStatus,
    Active,
    Inactive
};

enum car_class{
    NoClass,
    Basic,
    Green,
    Premium,
};
```

Fig 7 - *Enum* do **Account\_Status** & da **Car\_Class**

---

### 3 Queries

Após a estruturação do trabalho, e como objetivo definido no enunciado do projeto, criámos uma série de *queries* com o propósito de responder a determinadas questões propostas de modo a avaliar e validar o funcionamento e a eficiência do armazenamento e gestão da informação em memória. Nesta primeira fase do trabalho decidímos desenvolver as 3 primeiras *queries* propostas.

#### 3.1 Query I

**Descrição:** listar o resumo de um perfil registado no serviço através do seu identificador, representado por  $\langle ID \rangle$ . Note que o identificador poderá corresponder a um utilizador (i.e., *username* no ficheiro *users.csv*) ou a um condutor (i.e., *id* no ficheiro *drivers.csv*). Em cada caso, o *output* será diferente, mais especificamente:

- Utilizador  
nome;genero;idade;avaliacao\_media;numero\_viagens;total\_gasto
- Condutor  
nome;genero;idade;avaliacao\_media;numero\_viagens;total\_auferido

Caso o utilizador/condutor não tenha nenhuma viagens associadas, considerar a *avaliacao\_media*, *numero\_viagens*, e *total\_gasto* como sendo 0.000, 0, e 0.000, respetivamente.

**Resolução:** para obtermos os resultados esperados nesta primeira *query*, começamos por verificar se se trata de um *user* ou *driver*, analisando se a informação dada pelo comando é constituída por apenas números(*id*) ou por se contém letras(*username*). A partir daqui, usamos a função da *glib g\_hash\_table\_lookup*, encontramos o *User/Driver* na *hashtable* do catálogo correspondente através do seu identificador. Após guardarmos todas as informações necessárias no formato anteriormente definido, resta-nos apenas imprimir estas informações no ficheiro de *output*.

#### 3.2 Query II

**Descrição:** listar os N condutores com maior avaliação média. Em caso de empate, o resultado deverá ser ordenado de forma a que os condutores com a viagem mais recente surjam primeiro. Caso haja novo empate, deverá ser usado o *id* do condutor para desempatar (por ordem crescente). O parâmetro N é representado por  $\langle N \rangle$ .

**Resolução:** para a resolução desta *query*, começamos por analisar o comando dado para a inicializar e, a partir dele, retiramos o valor de N necessário para a execução da *query*. Após isto, criamos um array de tamanho N que será utilizado para guardar os top N *Drivers* por ordem crescente das características necessárias. A partir daqui, resta-nos apenas usar a estrutura *HashTableIter* dada pela *Library glib* que nos permite iterar pela *HashTable* onde guardamos o catálogo de *Drivers*. À medida que vamos percorrendo a *HashTable* vamos inserindo os *drivers* no array, respeitando os devidos parâmetros pré-definidos no enunciado. Por fim, acabamos por guardar as informações necessárias no formato definido num array de strings para depois as imprimir-mos no ficheiro de *output*.

---

### 3.3 *Query III*

**Descrição:** listar os N utilizadores com maior distância viajada. Em caso de empate, o resultado deverá ser ordenado de forma a que os utilizadores com a viagem mais recente surjam primeiro. Caso haja novo empate, deverá ser usado o *username* do utilizador para desempatar (por ordem crescente). O parâmetro N é representado por <N>.

**Resolução:** tendo em conta a semelhança desta *query* e da anterior, o método utilizado para a resolver foi também idêntico, sendo as únicas diferenças que o top N é agora um top de *Users*, logo vamos iterar sobre o catálogo de *Users* ao invés do catálogo de *Drivers*, e a ordem pela qual este top está ordenado não depende da avaliação média mas sim da distância total percorrida, sendo que esta alteração não altera a lógica por detrás da resolução.

---

## 4 Testes Funcionais

Para testar os resultados que obtemos e verificar o bom funcionamento do nosso projeto, criámos um executável de testes (*programa\_testes.c*) que funciona de forma idêntica ao programa principal, com a diferença de que compara os resultados obtidos aos resultados esperados (fornecidos pela equipa docente da UC), imprimindo se são ou não iguais no terminal. Para além disto, imprime também o tempo que o programa demora a executar a *query* pedida.

## 5 Encapsulamento e Modularização

Dada a grande importância dos conceitos de encapsulamento e modularização denotada no enunciado do projeto, estes foram aspectos sobre os quais tivemos grande atenção e cuidado. Em relação ao encapsulamento, fizemos questão de duplicar e utilizar a versão duplicada dos dados ao longo de todo o projeto de forma a não alterar o conteúdo original, além de termos definido todas as *structs* necessárias nos ficheiros ".c". Quanto à modularização, separámos o código em diferentes ficheiros de maneira a tornar a sua estruturação e visualização mais claras, bem como obter uma maior segurança e permitir a reutilização de código ao longo do trabalho.

---

## 6 Conclusão

O grupo considera que a primeira fase do trabalho sucedeu de forma mais que satisfatória, tendo sido alcançados todos os objetivos propostos no enunciado do trabalho e executando as três primeiras *queries* num tempo bastante satisfatório. Para além disso, deixamos já preparado um sistema de catálogos capaz de suportar tamanhos muito superiores aos propostos nesta fase e que nos vai permitir implementar as restantes *queries* sem enfrentar muitos obstáculos. Esta fase permitiu-nos ainda uma melhor compreensão de conceitos como encapsulamento e modularização de dados, que pretendemos utilizar e explorar mais na próxima fase.