Relatório da 1ª fase de LI3 – Grupo 70

Objetivos da 1ª fase, Makefile e inicialização do programa(main)

Esta primeira fase do projeto é maioritariamente focado no Batch, na modulagem e no encapsulamento. Nesta situação, nós tivemos de implementar uma forma em que o programa, ao ser executado, além de o correr também tinha de ir buscar os ficheiros onde temos a *data* necessária (os users, os drivers e as rides) e o input que programa iria utilizar, basicamente o conceito de Batch. Implementamos também a modulagem e tentamos encapsular a *data* o máximo possível.

Para criarmos o executável que seria executado para correr o programa, fomos encarregues de programar um Makefile, que faria o programa mais fácil de executar e depois de executado, de limpar todos os ficheiros não mais necessários, utilizando o *make clean*. No nosso caso, a nossa Makefile é relativamente simples, utilizando as flags recomendadas e criando os .o de todos os nossos módulos, sendo eles o main.c, calculos.c, data.c, querys.c e queryscheck.c (mais para a frente falaremos da utilidade de cada módulo e motivo de os termos separado assim e como isto ajuda na modulagem). Além disso depois de criar todos os .o, compila-os todos num só, o programa-principal, que é o executável que iremos utilizar, conjunto com os paths para os ficheiros necessários.

Além de criarmos cada .o, no nosso caso, nós resolvemos criar a pasta Resultados na nossa Makefile, ou seja, a pasta onde os outputs depois vão ser colocados, pois não faria sentido essa pasta existir antes do programa ser inicializado sequer. Depois de termos os outputs todos e de termos verificado se estes estão corretos, ao fazermos o *make clean* além de eliminar todos os .o criados, também elimina os .txt da pasta de Resultados e depois elimina a pasta.

Depois de fazermos make, ao executarmos o programa damos a localização da pasta onde está a *data* e onde se encontra o input. Depois de recebermos cada ficheiro, guardamos numa variável em específico e depois abrimos cada file de cada vez. Ao abrirmos os files, inicialmente criamos um array quer para os users, quer para os drivers e quer para os rides(userarray, driverarray e ridearray, respetivamente). Cada um destes arrays será criado com um malloc e com um MAX_USER, MAX_DRIVER e MAX_RIDE inicialmente definidos no data.h, cada um com o número respetivo de linhas do ficheiro em questão, ou seja, 10000 (cem mil), 10000 (dez mil) e 1000000 (um milhão) de espaço. Depois fazemos Parsing dos dados numas structs criadas por nós, mas depois falaremos delas.

Depois do Parsing, abrimos o input e criamos um lugar onde vamos guardar o nome do output. Após termos aberto o input e preparado o lugar onde iremos guardar o resultado, lemos a primeira linha do ficheiro de input. Antes de tentarmos perceber qual será o resultado, indicamos o lugar onde ficará o resultado ("../trabalho-pratico/Resultados/") e com o *snprintf* podemos criar uma forma que de cada vez que lê uma linha nova uma variável chamada it aumenta por um e assim o nome do ficheiro será "commandX_output.txt" sendo esse X o valor que it tem naquele momento. Inicialmente, estávamos a mandar tudo para um output, mas graças à descoberta da função snprintf, tudo tornou-se mais fácil. Posteriormente, adicionamos o nome do ficheiro ao filepath e abrimos o ficheiro em questão, que irá guardar o resultado. Copiamos a primeira linha do ficheiro de input para uma variável à parte para podermos utilizar a função queryid(a função que identifica a query que iremos utilizar), sem estragar a linha original, e depois enviamos tudo para a handle: os arrays com os dados já depois de ter sido feito o Parse, o queryid para sabermos qual query iremos utilizar, a linha do ficheiro que está a ser utilizada neste momento e o ficheiro de output.

Parsing dos dados

Um dos módulos mais importantes e mais básicos do nosso trabalho é o data.c. Este é o módulo que trata do Parsing do dados inteiro e de grande parte do nosso encapsulamento. No seu data.h é onde definimos a dimensão que será utilizado nos ficheiros, tal como mencionamos anteriormente. Além disso, é nele que estão definidas as structs que os users, os drivers e as rides vão utilizar durante o programa todo.

Antes de colocarmos cada variável no seu local, é no data.c que estão definidas as funções de alocar espaço: new_userarray, new_driverarray e new_ridearray.

Visto que nos ficheiros .csv onde estão guardados os dados os ficheiros têm na primeira linha a sua organização, nós decidimos guardar cada user, driver e ride da mesma forma que está lá descrita.

Sendo assim, a struct dos users é: char* username, char* name, char gender, char* birthdate, char* accountcreation, char* paymethod e char* account status.

Com esta informação, fizemos um parseusers, começando este por pegar numa linha do ficheiro, criar espaço para um user, cria uma variável context, para sabermos sempre em que parte do user estamos, e regista a primeira parte da linha até ao ";" como token, porque como em cada linha do ficheiro dos users.csv, cada informação do user está separada por ponto e

vírgula, sempre que encontra um, para e guarda essa informação no seu respetivo local, sendo este local dado pelo valor do context, que depois de ter sido colocado no sítio certo aumenta por um, sempre assim até ao accountstatus. Por exemplo: **MiTeixeira;Miguel Teixeira;M;03/09/1958;05/11/2017;cash;active**, como o ponto e vírgula aparece no fim do MiTeixeira, o username registado fica esse e o context passa para dois. Quando chega ao último e não encontra outro ";", reconhece que já está preenchido um user, logo passa para a próxima linha do ficheiro, faz malloc para um novo user e mete o context de novo a um para começar novamente no username. Esta informação está sempre a ser guardada pelo strtok, usando o ";" como ponto de paragem e fazemos strdup para guardar a informação no seu respetivo local. Quando o fgets dá igual a zero, significa que já vimos o ficheiro todo e que já podemos ir para os drivers.

Procedemos de maneira similar para os drivers e para as rides, mas tivemos obviamente de modificar as structs.

A struct dos drivers é, sendo assim, muito semelhante em forma à dos users, mas ligeiramente diferente no tipo de informação pedida, pois necessita de dados mais específicos a um driver: int id, char* name, char* birthdate, char gender, char *car_class, char *licenseplate, char *city, char *accountcreation e char *accountstatus.

Na parsedrivers, funciona tudo da mesma forma que funciona a parseusers, mas novamente com um diferente pedido de dados, sendo neste caso pedido a classe do carro, a matricula e a cidade do condutor. Por exemplo: **000000000001;Luísa Almeida;19/09/1990;F;green;EV-54-07;Setúbal;05/07/2021;active**, neste caso a primeira informação a ser guardada é o 000000000001, sendo esta guardada no id.

No caso das rides, temos um comentário que nem sempre existe, logo, no nosso caso, simplesmente guarda NULL, se não existir.

A struct das rides é um bocadinho mais única que as outras, mas funciona do mesmo modo: int id, char *date, int driver, char *user, char *city, int distance, int score_user, int score_driver, double tip, char *comment.

Novamente, a parserides funciona de forma similar às outras duas, mas com um tipo de dados completamente diferente às outras duas, pois guarda todas as informações que uma ride necessita, desde a data que foi realizada, o driver que a fez, a cidade em que foi, a distância percorrida e os scores de cada um dos utilizadores do carro. Por exemplo: **000000000002;19/10/2019;000000002536;LoSousa98;Faro;1;4;2;5.0;Image outside north effect than though sport**, o 000000000002 fica guardado no id, sendo este id o id da viagem.

Encapsulamento, reutilização de funções e modulagem

Mantendo-nos no data.c, foi aqui que nos deparamos pela primeira vez com uma tentativa de encapsular o programa. Para tentarmos manter o programa o mais encapsulado possível, tentamos manter as informações que os parses calcularam, única e exclusivamente acessíveis a partir de get_X, sendo o X a variável que o programa quereria naquele momento.

Para isto criamos cerca de 19 funções, cada uma para um aspeto diferente duma das structs que o programa poderia necessitar naquele momento. Acontece que, devido ao facto de termos reutilizado funções, algo que tentamos fazer ao longo do projeto, algo muito observável em várias queries, pois utilizam funções de outras queries e também structs auxiliares em alguns momentos, se não tivéssemos reutilizado funções poderíamos ter acabado por fazer cerca de 26 funções para ajudar com o encapsulamento, mais 7 funções do que as que realmente fizemos. A maneira que fizemos foi que como certas structs têm certas características iguais, basta criar uma função e depois, dentro desta, criar uma maneira com que esta saiba qual dos arrays é que tem de chamar. Apesar de não ser um conceito que se consiga dizer muito além de "Faz com que o programa não tenha sempre tudo ao seu dispor, a toda a hora e a ser constantemente lido, fazendo assim com que o programa seja mais rápido" é uma das partes mais importantes do projeto, pois faz o tempo de execução reduzir drasticamente.

Além de usarmos o encapsulamento aqui, existe outro local onde podemos falar dele que é na handle da main.c. É nesta função que fazemos com que o programa vá para uma query em específico, ou seja se o queryid for 1 vai para a query1check, se for 2 vai para a query2, se for 3 vai para a query3 e sempre assim. O que é que acontece? Antes de falarmos no encapsulamento, dá para reparar que se o queryid for1 não vai diretamente para a função da query mesmo, pois antes tem de passar por um processo que se encontra na queryscheck.c, onde determina se vai para a query1user ou para a query1driver.

Temos aqui um principal exemplo da modulagem, em que temos um módulo à parte somente para o querycheck da 1 e só depois de fazer esse check é que realmente entrar no ficheiro onde está o código das queries, o querys.c. Nós, em termos de modulagem, criamos cinco módulos:

- o módulo da main.c – onde tem a abertura dos ficheiros, a leitura do input, a criação do output e a handle para saber para que query é que vai ser utilizada

- o módulo data.c – onde está o Parsing dos dados, as structs nas quais colocamos os ficheiros e a maior parte do encapsulamento
- o módulo queryscheck.c – que atualmente somente serve para verificar para qual das queries a query1 vai
- o módulo querys.c – onde estão definidas todas as funções das queries que nós já implementamos
- o módulo calculos.c – onde estão as funções que as queries podem necessitar para calcular certas características que não se encontram na struct inicial, por exemplo, a avaliação média, que usamos na query 1 e 2.

Estes foram os módulos que nós achamos que precisávamos de criar para correr o nosso programa da maneira mais eficiente possível, pois sendo assim cada parte importante do código encontra-se num módulo diferente.

Voltando novamente ao encapsulamento e à função handle, dá para reparar que apesar de esta receber todos os arrays dos ficheiros, esta somente manda para a query em questão o array que esta necessita, sendo assim só a query1 é que recebe todos os arrays, por exemplo, a query6 só necessita das rides, pois só verifica rides, não precisa de saber quem é o user nem o driver.

Este foi o encapsulamento e a modulagem que nós conseguimos implementar, pensamos em meter cada query num módulo diferente, mas depois decidimos não o fazer, pois também achamos que seria algo em excesso e chegava a um ponto que o desempenho não melhorava assim tanto.

Queries implementadas

Query 1(Resumo de um perfil registado)

Antes de realmente entrarmos na query1, temos de utilizar a função previamente mencionada, a query1check. Caso tenhamos um int, sabemos que estamos na presença de um driver, logo vamos para a query1driver, e, neste caso, temos um pouco de encapsulamento pois já não necessitamos de enviar o userarray. Caso contrário, temos de ir para a query1user e estamos na presença da única query que necessita de todos os arrays.

A parte mais importante nesta query e na 2,3,7 e 8, apesar de algo fácil de verificar, é verificar inicialmente se não nos encontramos na presença de uma conta inativa, pois neste caso, se isso for verdade, não devolvemos nada e o output fica vazio com 0 bytes.

Depois de termos visto que a conta está ativa, primeiro, em ambos os casos, vamos buscar o nome, o gender e a idade, sendo que a idade é calculada com a função que se encontra em calculos.c, sendo que a data de

referência é 9/10/2022, estando esta definida com defines no início da calculos.h.

Após termos estas 3 características, temos de calcular a avaliação média de cada um, utilizando a função av_mediauser ou av_mediadriver, dependendo se é user ou driver, pois quer o user quer o driver, tem o seu próprio score.

Depois calculamos o número de viagens, outra vez com uma função num_viagensuser ou num_viagensdriver, correndo estas o ridearray todo vendo se o user ou o id do driver corresponde.

A última característica é o total gasto ou o total auferido, que já são um bocado mais complicadas. Enquanto que no total auferido, só temos de ver a car_class uma vez, pois é sempre o mesmo condutor, no total gasto, temos de estar sempre a ver quem é o condutor, ver a sua car_class e adicionar ao total atual. Além disso, não nos podemos esquecer das tips.

No final, com as características todas calculadas e identificadas, mandamos para o output em forma de: nome;género;idade;avaliação média;número de viagens; total gasto/auferido.

Query 2(Top N condutores com maior avaliação média)

Nesta query utilizamos uma struct auxiliar para melhorar o desempenho e para guardarmos melhor os dados.

Criamos espaço para um array de objetos da nova struct e vemos quantos N condutores são.

Vemos o id de todos os condutores e a sua account_status aos pares e guardamos também o seu nome na struct.

Depois chamamos a função av_media_data_recente, para este calcular a avaliação média e a data mais recente de cada um dos drivers, fazendo isto ao pares para melhorar o desempenho.

Depois retiramos todas as contas inactive deste array.

Para finalizar, recorremos a um insertion sort para organizar esse array, usando como comparador inicial, a avaliação média, se esta for igual, vem primeiro a data mais recente (por isso é que guardamos inicialmente na struct), depois, caso ambas as datas forem iguais, aí organizamos por id por ordem crescente.

No output, damos print aos N condutores, sendo a estrutura:

id;nome; avaliação média

id;nome; avaliação média

…

Query 3(N utilizadores com maior distância viajada)

Tentamos implementá-la, mas devido ao facto que são 100 mil utilizadores e temos de correr 1 milhão de viagens de cada vez, ainda não conseguimos implementar esta query de forma otimizada, demorando atualmente muito tempo a processar.

Query 4(Preço médio das viagens numa determinada cidade)/ Query 5(Preço médio das viagens num dado intervalo de tempo)

Nas queries 4 e 5, para reutilizar uma função, utilizamos a mesma função duas vezes, a função preco_medio. Enquanto que na query 4, enviamos uma cidade e depois pegamos nas rides todas dessa cidade, enviamos para a precoviagem, vemos qual é o driver, calculamos o seu preço e depois dividimos pelas viagens todas, na query 5, enviamos duas datas e vemos todas as rides que se encontram entre essas datas, enviamos novamente para a preco_medio utilizando o mesmo método, dividir pelo número de viagens e obtemos novamente o preço médio.

Quando enviamos as duas datas, temos de utilizar sempre a data_comparison, que se encontra no calculos.c, para termos a certeza que a ride em questão está dentro das datas definidas pela query.

Temos aqui novamente outro caso de reutilização de funções, quer na preco_medio que é usada em duas queries, quer na data_comparison que iremos usar na query6.

O output seria do modo: preco_medio.

Query 6(Distância média percorrida numa cidade num intervalo de tempo)

Na query 6, utilizamos novamente a data_comparison nas rides para irmos buscar todas as rides que se encontram entre essas datas, pegamos na distância percorrida, dividimos pelo número de viagens e obtemos o resultado necessário.

O output seria do modo: distancia_media.

Query 7(Top N condutores numa cidade por avaliação média)

Na query7 utilizamos outra struct auxiliar, onde guardamos todas as viagens da cidade pedida e os respetivo id da viagem, para facilitar cálculos. Depois, vendo as viagens de cada condutor nessa cidade, calculamos a sua avaliação média lá. Depois de termos calculado a sua avaliação média naquela cidade, fizemos insertion sort e em caso de empate, organizamos por id de forma decrescente.

Conclusão

Nesta primeira fase do projeto, conseguimos implementar a Batch de forma eficiente, o patching de dados funcionou de forma bem eficiente, a nossa modulagem encontra-se bem formulada, dividindo o código da melhor forma possível e o nosso encapsulamento foi feito da melhor maneira que encontramos no data.c.