

CURSO de Engenharia Informática

Lógica computacional 2017/2018

Elaborado por: Daniel Barreiro nº: 170221079 **Turma: 3º INF**

David Afonso nº:

Basket score

Índice

Conteúdo

[Introdução 2](#_Toc511074225)

[Conclusão 3](#_Toc511074226)

Introdução

Estruturas e implementações

* Organização do projeto

A estruturação e implementação das funções apresentadas no enunciado do projeto foram feitas tendo em conta os princípios de **modularidade** estudados nas aulas teóricas. De modo a tornar o código mais organizado e percetível foi tomada a decisão de criar bibliotecas customizadas para cada grupo de comandos assim como para cada tipo de registos contidos nos ficheiros .csv (Jogos e Jogadores). A organização dos ficheiros do projeto é a seguinte:

|  |  |
| --- | --- |
| Ficheiros | Descrição |
| Players | Estruturas e funcionalidades para a criação e manipulação dos dados dos jogadores. |
| Games | Estruturas e funcionalidades para a criação e manipulação dos dados dos jogos. |
| PlayersInfo | Funcionalidades do Grupo A de comandos (**LOADP, SHOWP, TABLE, SEARCHP**) |
| PlayersPerformance | Funcionalidades do Grupo B (**LOADG, SEARCHG, MVP**) |
| GamesInfo | Funcionalidades do Grupo C (**MFOULP, MFOULG, FAIRP**) |
| AdvanceIndicators | Funcionalidades do Grupo D (**IDEALTEAM**) |
| FileHandler | Funcionalidades de *parsing* e importação dos ficheiros csv tanto para jogadores como para jogos. |
| UserInterface | Funcionalidade de leitura de comandos e apresentação do UI. |

As dependências foram organizadas de modo a que as bibliotecas Players e Games contendo as estruturas e funcionalidades necessárias para manipular os dados carregados fossem centrais a todas as outras bibliotecas.

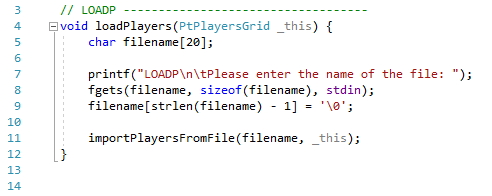
Grupo A

As funcionalidades do Grupo A segundo o enunciado “Informação sobre os Jogadores”, encontram-se na biblioteca **PlayersInfo**.

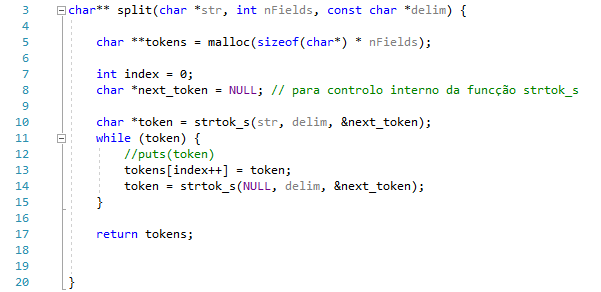
* Loadp

*“Pede o nome dum ficheiro, abre o ficheiro de Jogadores carrega-o em memória, se o ficheiro não puder ser aberto, escreve "FICHEIRO INACESSIVEL" e estruturas de dados ficam vazios. Se o ficheiro não for do tipo Jogadores escreve “FICHEIRO INCORRETO”.”*

Após a realização do documento de ajuda ao projeto a funcionalidade de abrir e ler a informação contida nos ficheiros do tipo **player\_n.csv**, faltou apenas criar as estruturas para armazenar a informação e funções necessárias para adicionar às mesmas.

A função **loadPlayers**, a responsável pela funcionalidade, pede ao utilizador um nome de ficheiro a ser carregado e chama a função **importPlayersFromFile** presente na biblioteca **FileHandler** para a importação dos dados.

A função **importPlayersFromFile**, fornecida na ficha de apoio ao projeto, abre o ficheiro csv e itera cada linha do ficheiro até ao seu fim. Em cada linha a informação é guardada numa string que é decomposta nos campos individuais (separados por vírgulas) pela função **split** e guardada num array de strings (tokens).



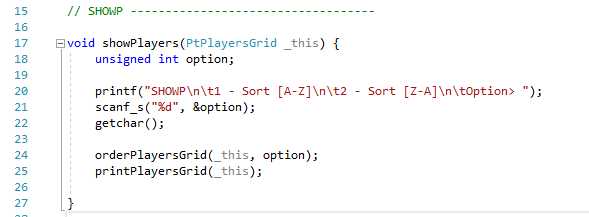


Após a iteração de cada linha a informação armazenada no array de strings (tokens) é enviada para a função **createPlayer** onde é criado um registo da estrutura **Player** que por si é adicionado ao array de jogadores da estrutura **PlayersGrid**.

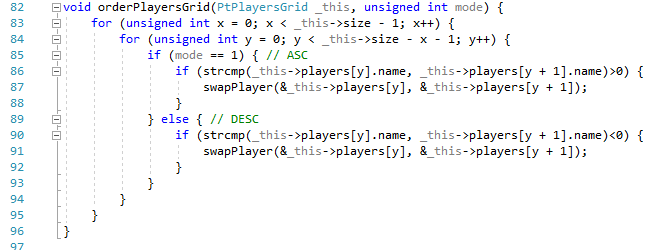
* SHOWP

*“Pergunta se a ordenação é de (A-Z) ou de (Z-A). Mostra todos os jogadores ordenados por nome, um por linha, indicando o seu id, em que equipa jogam e a data de nascimento.”*

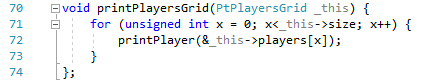
A função **showPlayers**, pede ao utilizador um método de ordenação (1:[A-Z], 2:[Z-A]) e utiliza as funções **orderPlayersGrid** e **printPlayersGrid** da biblioteca **Players**, para a ordenação dos registos dos jogadores por ordem alfabética escolhida e apresentação das informações no ecrã respetivamente.

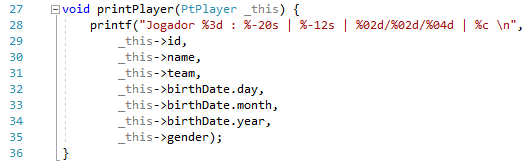


A função **orderPlayersGrid** utiliza o algoritmo **selection sort** estudados nas aulas, tendo em conta o método de ordenação escolhido pelo utilizador.



A função **printPlayersGrid** simplesmente itera cada registo do array de jogadores guardado na estrutura **PlayersGrid** e utiliza o método **printPlayer** da biblioteca **Players** para apresentar a informação do jogador.

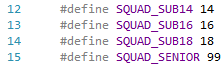




* TABLE

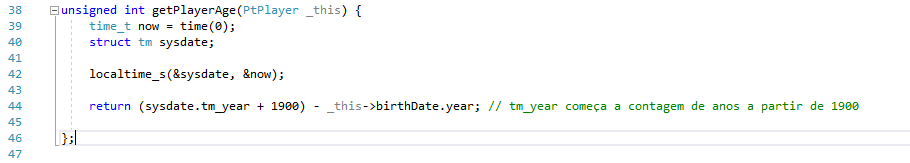
*“Mostra da chamada "Table Squads". Trata-se de uma tabela (matrix 4x2) que mostra quantos jogadores existem por escalão etário e por género”*

Utilizando constantes definimos os as idades dos escalões definidos no enunciado na biblioteca **PlayersInfo**, uma vês que o escalão sénior não tem um limite de idade definimos 99 como o valor da idade mínima desse escalão.



Após os escalões estarem definidos, criou-se a função para calcular a idade de cada jogadora através da sua data de nascimento. A função **getPlayerAge** da biblioteca **Players**, recebe o registo de um jogador e utilizando o comando **localtime\_s** retorna o ano atual, ao subtrair o ano atual com o ano da data de nascimento do utilizador obtemos a sua idade.

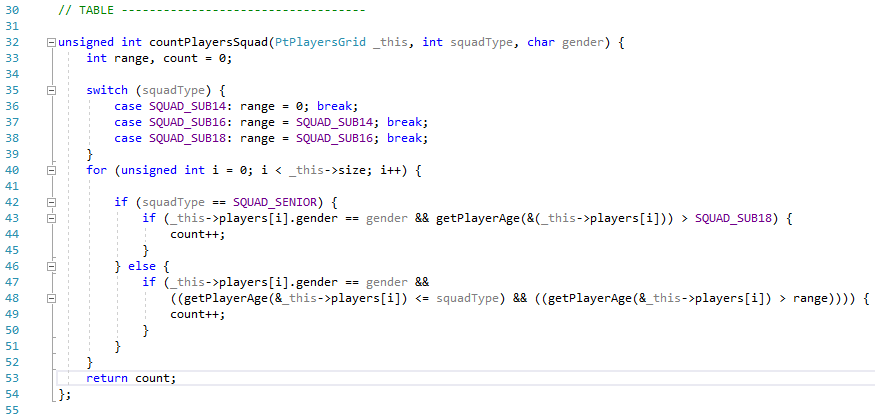
**Nota:** Esta maneira não tem em conta o dia e mês de nascimento, como tal se um sendo a data atual 15/04/2018 um jogador com 22 anos que faça anos em 16/04/2018 terá 23 anos mesmo ainda não ter chegado o seu dia de anos, mas uma vez que os escalões de idades têm em conta o ano e não o dia e o mês esta solução é válida!



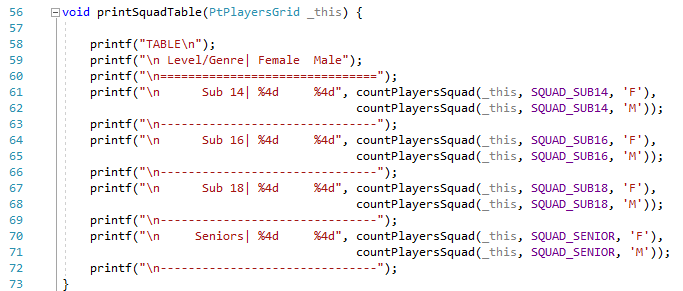
Com todas as funcionalidades criadas para obter a informação necessária para a funcionalidade pedida criou-se a função **countPlayersSquad** que recebendo o *array* de jogadores, um escalão e um sexo (Masculino ou Feminino) retorna a contagem de jogadores que respeitem o sexo enviado e a sua idade esteja dentro dos limites do escalão.

Em primeiro lugar é definido um *range* entre a idade mínima do escalão e a idade máxima, através do *switch case* podemos ver que tirando o escalão sub14 que não tem uma idade mínima e o escalão sénior, todos os outros escalões utilizam o escalão inferior como idade mínima.

De seguida é percorrido o array de jogadores em que a cada iteração é feita a verificação do sexo do jogador e, caso o tipo de escalão seja sénior, se a idade for superior à idade mínima do escalão sub18, caso o tipo de escalão seja outro, se a idade estiver entre a idade mínima e máxima do escalão, incrementando o contador a cada jogador válida. No fim retornar o contador calculado.



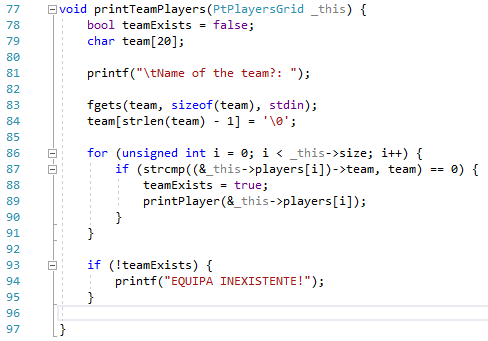
Por fim a função **printSquadTable**, escreve no ecrã a estrutura da tabela e por cada linha de informação invoca a função **countPlayersSquad** com o escalão e o sexo desejado.



* SEARCH

*“Pede o nome de uma equipa e mostra todos os jogadores dessa equipa. Se não existir deve mostrar “EQUIPA INEXISTENTE”.”*

A função **printTeamPlayers,** pede o nome de uma equipa ao utilizador, percorre o *array* de jogadores e por cada jogadora compara o nome da sua equipa com a equipa introduzida pelo utilizador, caso os dois nomes sejam idênticos, apresenta a informação do jogador utilizando da função **printPlayer** e coloca a variável **teamExists**, declarada no inicio como *false (bool)*, a *true*. No fim de percorrer o array caso a variável **teamExists**, continue com estado false apresenta a mensagem “EQUIPA INEXISTENTE!”



Grupo b

As funcionalidades do Grupo B segundo o enunciado “Informação especifica sobre o desempenho dos jogadores durante os jogos”, encontram-se na biblioteca **PlayersPerformanceInfo**.

* LOADG

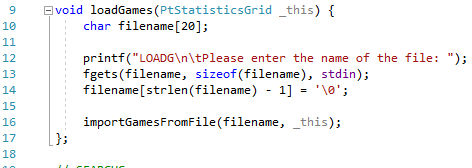
*“Pede o nome dum ficheiro, abre o ficheiro de Jogos carrega-os em memória. Se o ficheiro não puder ser aberto, escreve "FICHEIRO INACESSIVEL" e as estruturas de dados ficam vazias. Se o ficheiro não tiver o formato estabelecido em 1.2 para “games” escreve “FICHEIRO INCORRETO”.”*

Similarmente à função LOADP, criou-se um conjunto de estruturas **statistics**, **playerGameStatistics** fornecidas no enunciado do projeto para armazenar os dados as informações de cado jogo e **statisticsGrid** para armazenar o array de todos os jogos contidos no ficheiro.

Na biblioteca **FileHandler** foi criada a função **importGamesFromFile**, que tal como a função **importPlayersFromFile**, abre o ficheiro .csv, itera por cada linha, guardando a informação na estrutura **playersGameStatistics**, e adicionado ao array de jogos **PlayerGamesStatistics**.



A função principal **loadGames**, pede ao utilizador o nome de um ficheiro para carregar, e de seguida invoca a função **importGamesFromFile** para efetuar o carregamento da informação e/ou validar se ficheiro existe.



* SEARCHG

*“Pede o id do Jogo, caso não existe nenhum registo para o id dado, escreve “JOGO INEXISTENTE”, senão: mostra para o jogo identificado:*

* *Numero total de pontos marcados por jogo;*
* *Numero de jogadores utilizados por jogo;*
* *Numero de bloqueios efetuados por jogo;”*

Muito semelhante à funcionalidade SEARCHP, a função pede ao utilizador um ID de um jogo, e de seguida percorre o o ***array*** dos jogos, caso encontre algum registo com o mesmo id, coloca a variável **gameExists** (bool), declarada a false no início, a true, e incrementa as variáveis **totalPoints** (multiplicando o número de 3 pontos por 3, e o número de 2 pontos por 2), **totalBlock** e **totalPlayers**. No final se a variável **gameExists** for *true*, apresenta os resultados calculados no ecrã, caso a variável for false, apresenta a mensagem “JOGO INEXISTENTE”.



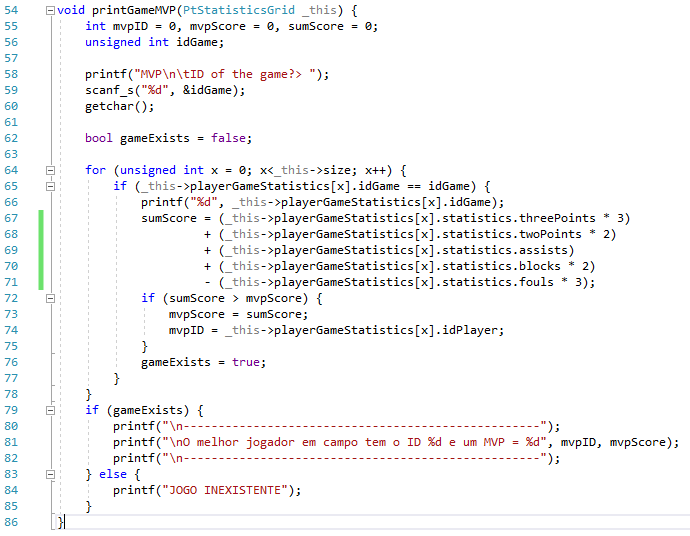
* MVP

*“Pede o id do Jogo, caso não existe nenhum registo para o id dado, escreve “JOGO INEXISTENTE”, senão para o Jogo em questão identifica o melhor jogador em campo e indica o seu índice de MVP. O melhor jogador em campo é aquele com melhor índice de MVP o jogador – (most valuable player) que é calculado pela seguinte formula:*

*MVP = 3 x tree\_points + 2 x two\_points + assists + 2 x blocks – 3 x fouls.”*

Tal como o comando anterior, a função **printGameMVP**, pede ao utilizador um id de um jogo, em seguida, para cada jogador calcula utilizando a fórmula MVP os pontos ganhos no jogo desejado e armazena o id do jogador na variável **mvpID** e o valor calculado na variável **mvpScore**, caso outro jogador obtenha um resultado melhor que ao previamente armazenado na variável **mvpScore**, então substituído o respetivo valor e id, a variável **gameExists** também é colocada a true a cada iteração.

No final do ciclo, caso a variável **gameExists** for true apresenta os id e os resultados do melhor jogador do jogo escolhido, caso contrário apresenta a mensagem “JOGO INEXISTENTE”.



Grupo C

As funcionalidades do Grupo B segundo o enunciado “Informação agregada sobre jogos e jogadores”, encontram-se na biblioteca **GamesInfo**.

MFOULP

*“Mostra a média de faltas dadas por jogador por jogo (Exemplo: Tim Smiths deu em media 1.5 faltas por jogo).”*

A função **printStatisticsFouls**, recebe o array de jogadores e o array de jogos e por cada jogador, itera por todos os jogos contido no array de jogos e incremente as variáveis **numberOfGames** e **numberOfFouls,**.com o número de faltas do jogador. No final verifica se o jogador teve algum jogo ou falta, se sim imprime os resultados obtidos, se o jogador participou em jogos mas não teve faltas apenas imprime o número de jogos em que participou, caso o jogador não tenha nenhum jogo nem falta não é apresentado.



MFOULG

*“Mostra a média de faltas cometida por jogo por jogador. (Exemplo: No jogo com*

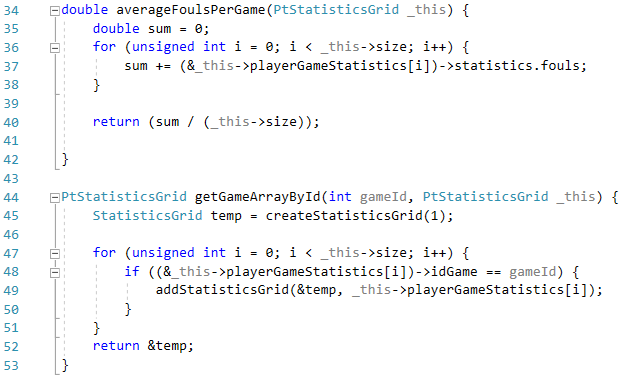
*id=3 em media cada jogador deu 0.8 faltas)*

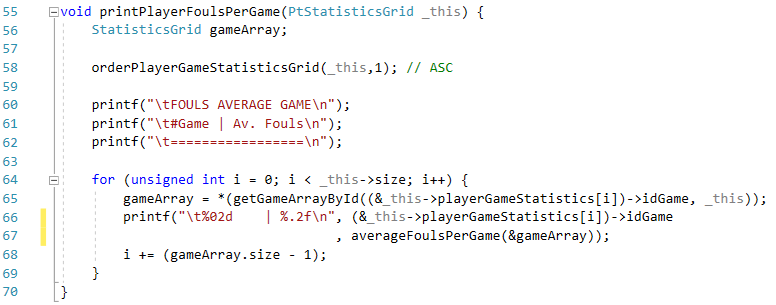
*Nota: Sugere-se que comece por determinar qual a gama de valores do idGame,*

*de forma a “balizar a sua pesquisa), e em seguida utilize um array auxiliar para*

*guardar o número de faltas por jogo, que deverá inicializar a zero.”*

Utilizando a sugestão fornecida no enunciado, criou-se a função **getGameArrayById** que, fornecendo um ID de um jogo e o array de jogos importado, devolve um array (do tipo **StatisticsGrid**) com todos os registos associados ao id do jogo fornecido. Criou-se também a função **averageFoulsPerGame**, que recebendo o array gerado na função anterior calcula a média de faltas.





FAIRP

Conclusão