

CURSO de Engenharia Informática

Algoritmos e tipos abstratos de dados 2017/2018

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

David Afonso nº: 170221081

Docente: Patrícia Macedo

Basket score

Índice

Conteúdo

[Introdução 2](#_Toc512285009)

[Organização do projeto 3](#_Toc512285010)

[Estruturas 4](#_Toc512285011)

[Date 4](#_Toc512285012)

[Player e PlayersGrid 4](#_Toc512285013)

[Statistics e PlayerGameStatistics 4](#_Toc512285014)

[Statisticsgrid 5](#_Toc512285015)

[Teaminfo e TeamInfoGrid 5](#_Toc512285016)

[Grupo A 6](#_Toc512285017)

[LOADP 6](#_Toc512285018)

[SHOWP 7](#_Toc512285019)

[TABLE 9](#_Toc512285020)

[SEARCH 10](#_Toc512285021)

[Grupo b 11](#_Toc512285022)

[LOADG 11](#_Toc512285023)

[SEARCHG 12](#_Toc512285024)

[MVP 13](#_Toc512285025)

[Grupo C 14](#_Toc512285026)

[MFOULP 14](#_Toc512285027)

[MFOULG 15](#_Toc512285028)

[FAIRPLAY 16](#_Toc512285029)

[Grupo D 20](#_Toc512285030)

[IDEALTEAM 20](#_Toc512285031)

[Conclusão 23](#_Toc512285032)

Introdução

Este projeto e relatório foram desenvolvidos no âmbito da unidade curricular Algoritmos e Tipos Abstratos de Dados com o objetivo de desenvolver e implementar um conjunto de funcionalidades pedidas no enunciado do projeto. O projeto insere-se no seguinte contexto (segundo o enunciado):

*“A Federação Portuguesa de Basquetebol (http://www.fpb) mantem um conjunto de informação detalhada sobre cada jogo (pontos, faltas, assistências e defesas etc. Os dados são recolhidos em cada jogo, e posteriormente tratados estatisticamente, sendo possível obter indicadores de desempenho por jogo e por jogador.”*

O objetivo deste projeto foi desenvolver um programa na linguagem C para extrair informação de ficheiros (do tipo .csv) sobre os jogadores e os jogos de basquetebol, utilizando os conhecimentos e técnicas adquiridos ao longo das aulas.

Para o desenvolvimento deste projeto foram utilizados os seguintes programas:

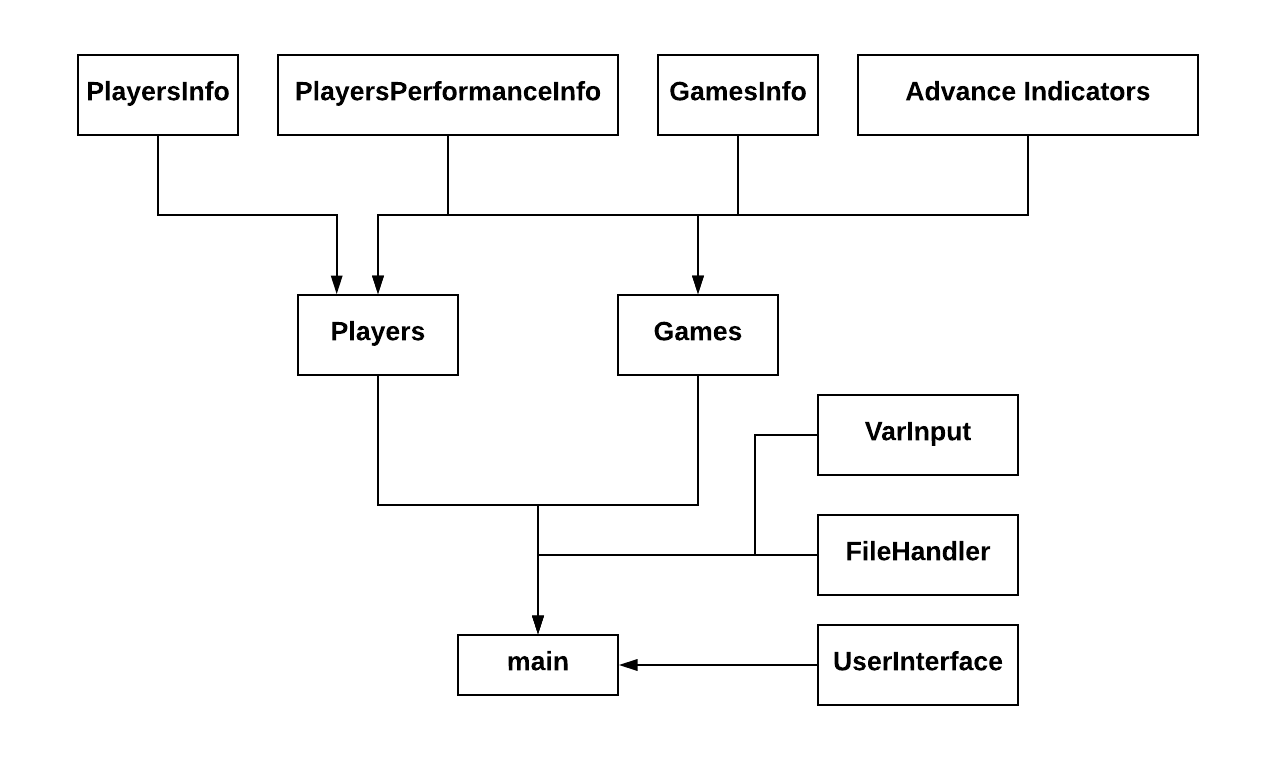
* Microsoft Visual Studio 2017
* Github

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. |

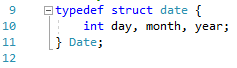
As dependências foram organizadas de modo a que as bibliotecas Players e Games contendo as estruturas e funcionalidades necessárias para manipular os dados carregados fossem centrais a todas as outras bibliotecas.



Estruturas

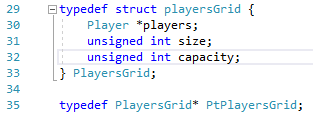
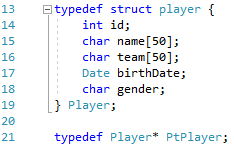
As estruturas utilizadas no projeto são na sua maioria as referidas no documento de apoio ao projeto, todas as outras estruturas foram desenvolvidas como forma de simplificar a complexidade do código de um dos comandos.

Date



A estrutura **Date** armazena informação da data de nascimento de um jogador, nomeadamente o dia, mês e ano da data.

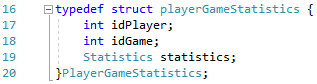
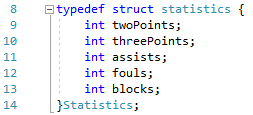
Player e PlayersGrid



A estrutura **Player** armazena a informação de cada jogador presente nos ficheiros relativos aos jogadore. Constituído por um id de jogador, nome, nome de equipa, data de nascimento (utilizando a estrutura date) e sexo (F para Feminino ou M para masculino).

A estrutura **PlayerGrid** armazena um array de jogadores, bem como o tamanho do array e a capacidade (inicialmente 200).

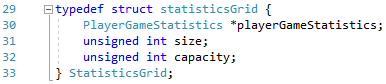
Statistics e PlayerGameStatistics



A estrutura **Statistics** armazena número de pontos, faltas, assistências e bloqueios de um jogador realizados durante um certo jogo.

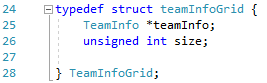
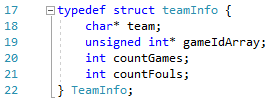
A estrutura **PlayerGameStatistics** armazena um array de estatísticas, o id do jogador ao qual as estatísticas pertencem e o id do jogo realizado.

Statisticsgrid



A estrutura **StatisticsGrid** armazena um array de um jogo de um jogador, o tamanha e a capacidade do array.

Teaminfo e TeamInfoGrid



A estrutura **TeamInfo** armazena as informações de uma equipa (utilizada no comando FAIRPLAY). Contêm o nome da equipa, um array com os ids dos jogos realizados pela equipa, o número de jogos e o número de faltas realizadas pelos jogadores da equipa.

A estrutura **TeamInfoGrid** armazena um array de informações das equipas existentes no ficheiro de jogos.

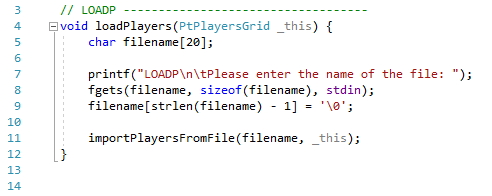
Grupo A

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

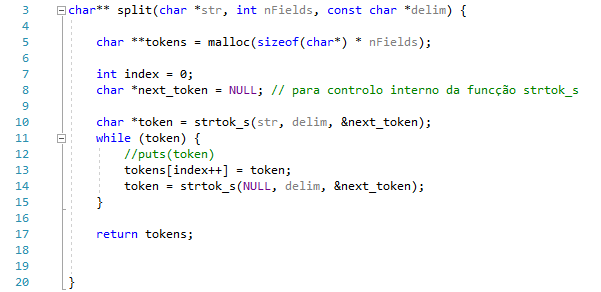
LOADP

* **Funções:** loadPlayers
* **Complexidade Algoritmica:**
* **Enunciado**: *“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.

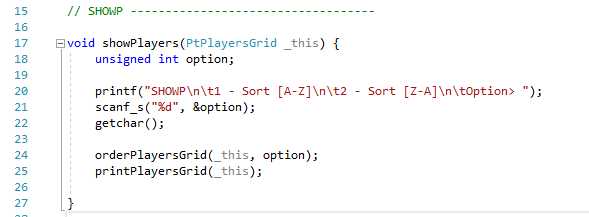
A função **importPlayersFromFile**, fornecida na ficha de apoio ao projeto, abre o ficheiro csv e itera cada linha do ficheiro até ao seu fim. Em cada linha a informação é guardada numa string que é decomposta nos campos individuais (separados por vírgulas) pela função **split** e guardada num array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

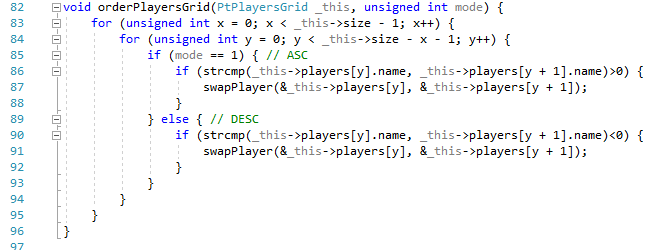




Após a iteração de cada linha a informação armazenada no array de strings (tokens) é enviada para a função **createPlayer** onde é criado um registo da estrutura **Player** que por si é adicionado ao array de jogadores da estrutura **PlayersGrid**.

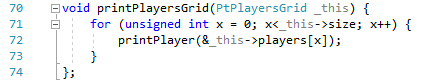
SHOWP

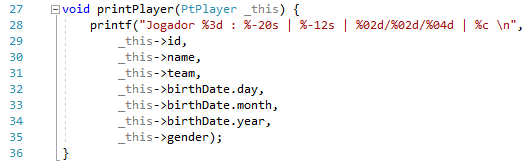
* **Funções:** showPlayers
* **Complexidade Algoritmica:**
* **Enunciado**: *“Pergunta se a ordenação é de (A-Z) ou de (Z-A). Mostra todos os jogadores ordenados por nome, um por linha, indicando o seu id, em que equipa jogam e a data de nascimento.”* 

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.

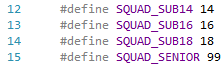




TABLE

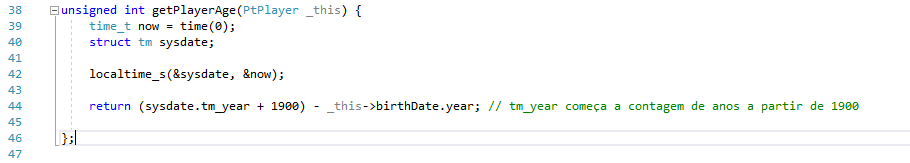
* **Funções:** getPlayerAge, countPlayersSquad, printSquadTable
* **Complexidade Algoritmica:**
* **Enunciado**: *“Mostra da chamada "Table Squads". Trata-se de uma tabela (matrix 4x2) que mostra quantos jogadores existem por escalão etário e por género”*

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.

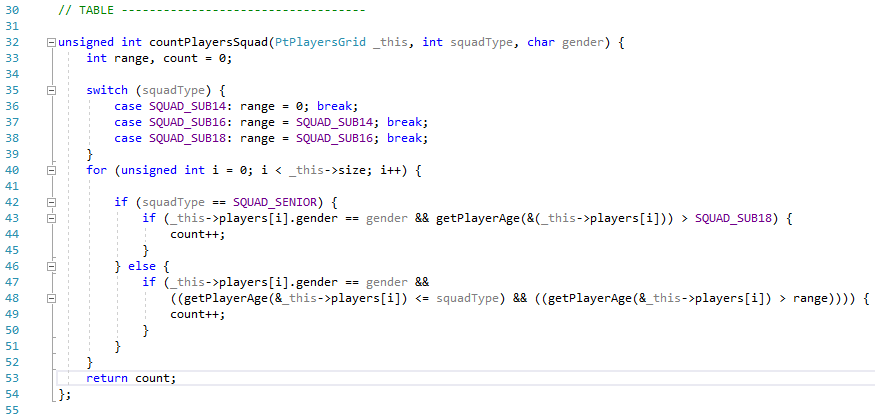
**Nota:** Esta maneira não tem em conta o dia e mês de nascimento, como tal se um sendo a data atual 15/04/2018 um jogador com 22 anos que faça anos em 16/04/2018 terá 23 anos mesmo ainda não ter chegado o seu dia de anos, mas uma vez que os escalões de idades têm em conta o ano e não o dia e o mês esta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



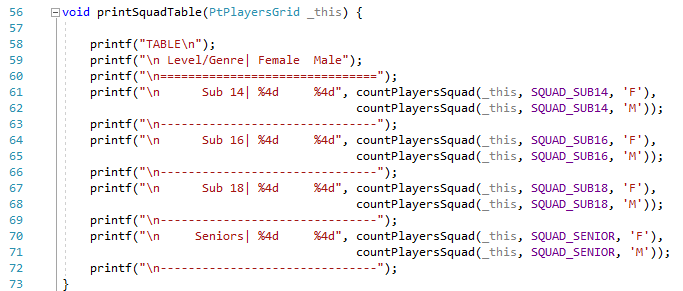
Com todas as funcionalidades criadas para obter a informação necessária para a funcionalidade pedida criou-se a função **countPlayersSquad** que recebendo o *array* de jogadores, um escalão e um sexo (Masculino ou Feminino) retorna a contagem de jogadores que respeitem o sexo enviado e a sua idade esteja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



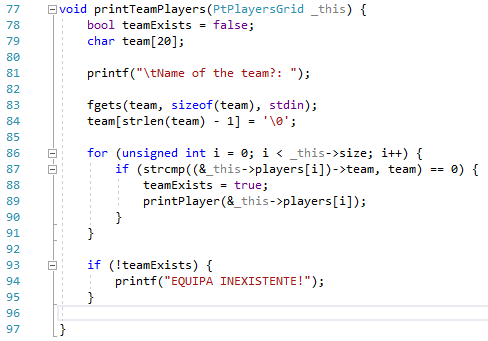
Por fim a função **printSquadTable**, escreve no ecrã a estrutura da tabela e por cada linha de informação invoca a função **countPlayersSquad** com o escalão e o sexo desejado.



SEARCH

* **Funções:** printTeamPlayers
* **Complexidade Algoritmica:**
* **Enunciado**:*“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

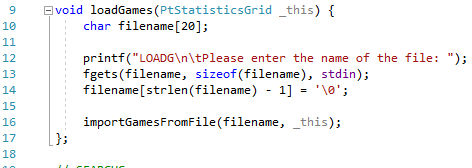
* **Funções:** loadGames
* **Complexidade Algoritmica:**
* **Enunciado**: *“Pede o nome dum ficheiro, abre o ficheiro de Jogos carrega-os em memória. Se o ficheiro não puder ser aberto, escreve "FICHEIRO INACESSIVEL" e as estruturas de dados ficam vazias. Se o ficheiro não tiver o formato estabelecido 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

* **Funções:** printGameGrid
* **Complexidade Algoritmica:**
* **Enunciado**: *“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”.



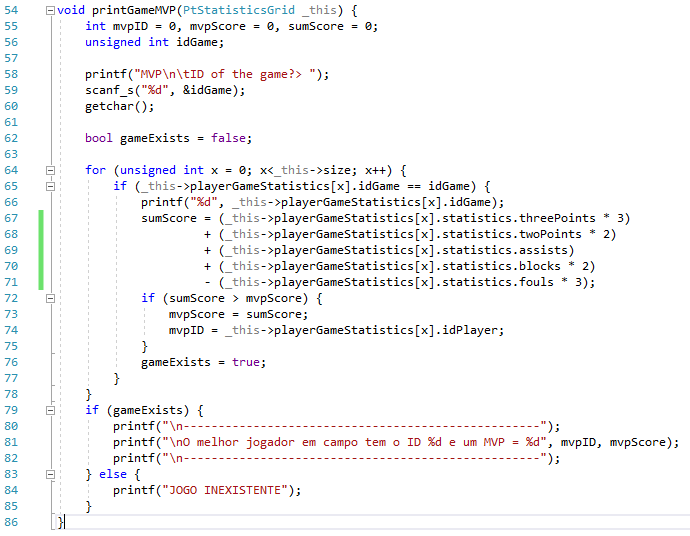
MVP

* **Funções:** printGameMVP
* **Complexidade Algoritmica:**
* **Enunciado**: *“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 ids 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

* **Funções:** printStatisticFouls
* **Complexidade Algoritmica:**
* **Enunciado**: *“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

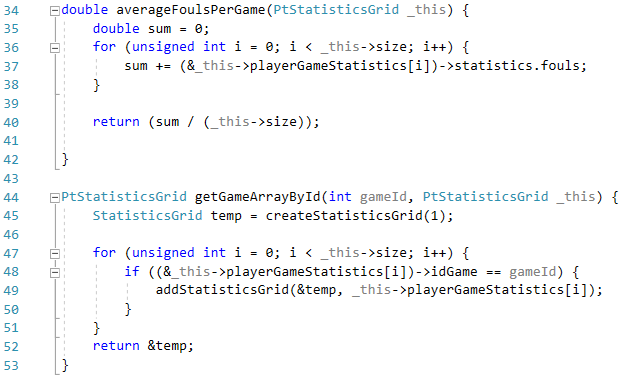
* **Funções:** getGameArrayById, averageFoulsPerGame, printPlayerFoulsPerGame
* **Complexidade Algoritmica:**
* **Enunciado**: *“Mostra a média de faltas cometida por jogo por jogador. (Exemplo: No jogo com id=3 em media cada jogador deu 0.8 faltas)*

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

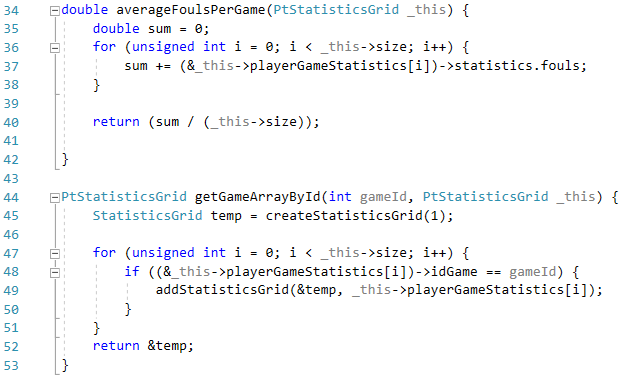
*de forma a “balizar a sua pesquisa), e em seguida utilize um array auxiliar para*

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

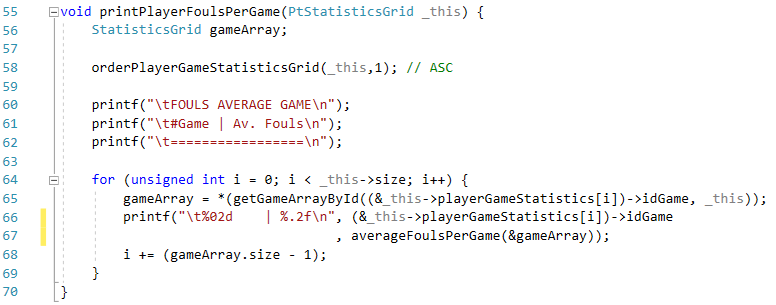
Utilizando a sugestão fornecida no enunciado, criou-se a função **getGameArrayById** que, fornecendo um ID de um jogo e o array de jogos importado, devolve um array (do tipo **StatisticsGrid**) com todos os registos associados ao id do jogo fornecido.



Criou-se também a função **averageFoulsPerGame**, que recebendo o array gerado na função anterior calcula a média de faltas.



Com estas duas funções anteriores, a função principal do comando, **printPlayersFoulsPerGame,** que recebe o array de jogos importados, cria um array temporário de jogos, ordena, por id, de forma ascendente o array recebido e de seguida realiza um ciclo por todos os registos do mesmo. Por cada iteração o comando utiliza a função **getGameArrayById**, para preencher o array temporário com todos os registos com o mesmo id. Após preencher o array temporário, utiliza-se o mesmo para calcular a média do jogo e apresentar no ecrã o resultado. Por fim incrementa o índex do ciclo com o tamanho do array temporário, uma vez que o array recebido está ordenado garante-se que na próxima iteração do ciclo principal o id de jogo será diferente.

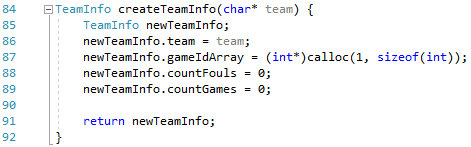


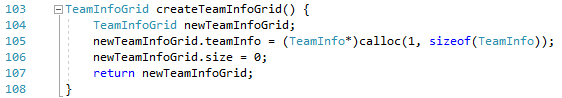
FAIRPLAY

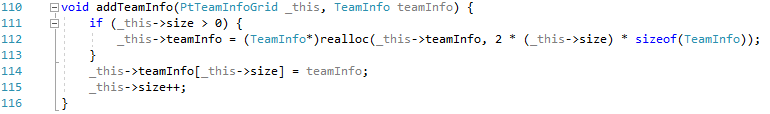
* **Funções:** getPlayerTeam, printFairPlayStatistics, orderFairPlayStatistics
* **Complexidade Algoritmica:**
* **Enunciado**: *“Listar as equipas em função do número médio de faltas por jogo. Nota: deverá determinar para cada equipa, quantos faltas ela cometeu, e quantos jogos jogou. E assim determinar a media de faltas por equipa por jogo. Em seguida deverá ordenar a lista crescentemente pelo número medio de faltas.”*

De forma a evitar um elevado número de ciclos entre os ficheiros de jogadores e os ficheiros de jogos, para este comando foi decidido montar uma estrutura que armazena-se, organiza-se a informação necessária através de um único clico e no final ordenar essa informação e apresenta-la no ecrã.

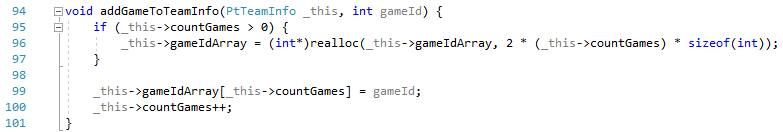
Criou-se as funções necessárias para criar e adicionar registos às duas estruturas **TeamInfo** e **TeamInfoGrid**, muito semelhantes às estruturas **PlayersInfo** e **StatisticsGrid**.



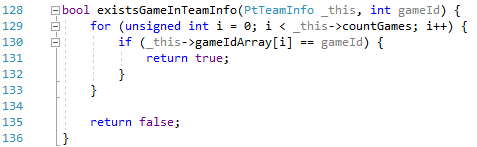




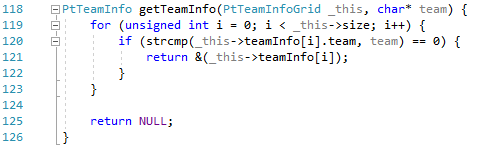
Para adicionar um id de um jogo ao array jogos realizados por uma equipa na estrutura **TeamInfo**, criou-se a função **addGameToTeamInfo,**



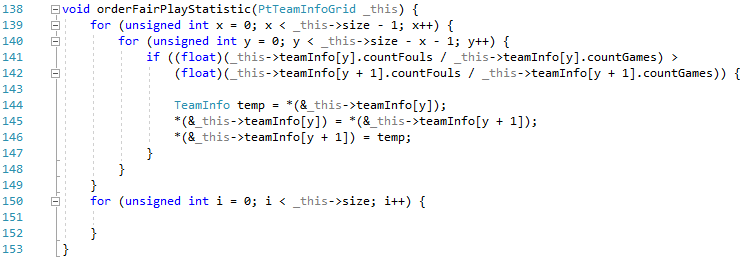
Uma vez que os ficheiros de jogos contêm Ids de jogos repetidos, criou-se a função **existsGameInTeamInfo** que recebe um id de um jogo e a informação de uma equipa (contêm o array de jogos realizados pela equipa) e verifica se a informação da equipa já contêm o registo desse jogo, se existir devolver true, senão false.



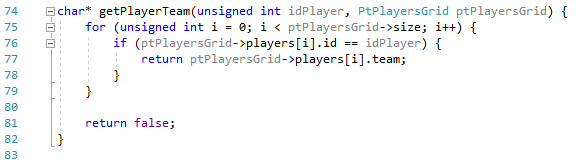
A função **getTeamInfo**, verifica e devolve a informação de uma equipa, recebe o array de informações de uma equipa e o nome de uma equipa, percorre o array recebido e procura um registo com o mesmo nome de equipa se sim devolve o endereço de memória do mesmo senão devolve NULL.



Para ordenar a informação como pedido no enunciado criou-se a função **orderFairPlayStatistic**, que recebe o array de informações de equipas e ordena-o de forma crescente pelo número médio de faltas.



Para obter a equipa de um jogador criou-se a função **getPlayerTeam** , recebe o id de um jogador e o array de jogadores importado. Procura pelo id de jogador e se existir retorna o nome da sua equipa, caso contrário retorna NULL.



Por fim a função **printFairPlayStatistic**, recebe o array de jogos importado, cria um array de informações de equipas e realiza um ciclo ao array recebido. Por cada iteração envia o id de jogador para a função **getPlayerTeam** e de seguida e verifica se a equipa do jogador já foi inserida no array de informações de equipas.

* Se a equipa já foi inserida: Verifica se o id do jogo já está registado na informação da equipa, se sim adiciona o id do jogo (**addGameToTeamInfo**) se não ignora o id repetido, de seguida incrementa a soma de faltas da equipa.
* Se a equipa ainda não foi inserida: Cria um novo registo **TeamInfo** e adiciona-o ao array de informações de equipa, adiciona o id do jogo à ao registo **TeamInfo** e inicializa o número de faltas da equipa.

No fim do ciclo é chamada a função **orderFairPlayStatistic** para organizar as informações das equipas crescentemente pela média de faltas e por fim é feito um último ciclo para apresentar as informações recolhidas no ecrã.



Grupo D

As funcionalidades do Grupo B segundo o enunciado “Indicador Avançados”, encontram-se na biblioteca **GamesInfo**.

IDEALTEAM

* **Funções:** getPlayerTeam, printFairPlayStatistics, orderFairPlayStatistics
* **Complexidade Algoritmica:**
* **Enunciado**: *“Pede o escalão da equipa ideal (0-sub14;1- sub16; 2-sub18; 3-senior). Pede o género (Feminino, M-Masculino), caso não existam 5 jogadores que preencham os critérios de escalão e género deverá escrever (“NÃO EXISTEM JOGADORES PARA A EQUIPA IDEAL), senão apresenta a equipa ideal, listando os dados dos 5 jogadores, indicando a função que assumem na equipa. A formação da equipa ideal deve satisfazer os seguintes critérios:*

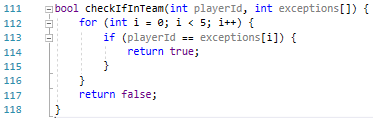
*1 jogadores do tipo CENTER – Jogadores que fazem mais assistências por jogo.*

*2 jogadores do tipo SHOOTY GUARDS – Jogadores que fazem mais pontos por jogo.*

*2 jogadores do tipo POINT GUARD” – Jogadores que fazem mais blocos(defesas) por jogo.*

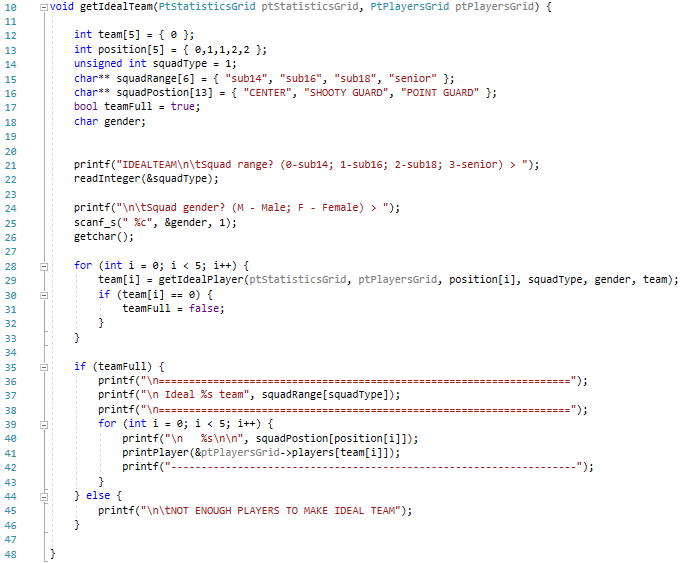
*Nota: Em caso de empate, deverão escolher o jogador que participou em mais jogos. Em caso de empate o Jogador mais novo.”*



**

A função **getIdealPlayer** calcula o jogador ideal para um dos tipos de jogadores selecionados, para tal é feito um ciclo ao array de jogadores e por cada jogador é feita outra iteração ao array de jogos para, dependendo do tipo de jogador a ser calculado, do escalão e do sexo, somar os valores necessários (assistências, pontos ou defesas) para o cálculo bem como o número de jogos que o jogador jogou, a função recebe também um array (exceptions) com ids de jogadores que já foram escolhidos e a cada iteração através da função **checkIfInTeam** verifica se o já é um dos escolhidos, se sim este jogador não será calculado.

Ao fim de cada iteração do array de jogadores é comparado com os valores do último jogador ideal guardado (caso exista), se a soma dos valores for superior ao do jogador ideal é trocado o jogador guardado, caso a soma dos valores seja igual e o numero de jogos calculado seja maior que a do jogador guardado é feito a troca, se a soma dos valores é igual e o número de jogos é igual é feita uma verificação à idade, se a idade do jogador calculado for menor que a idade do jogador guardado (mantendo-se dentro do escalão recebido pela função) é feita a substituição, caso contrário mantêm-se o jogador previamente calculado.



Por fim a função **getIdealTeam**, pede ao utilizador um escalão e um sexo, utilizando a função anterior calcula o número de jogadores pedido para cada tipo (total 5) e vai adicionando-os a um array de ids de jogadores, no final caso a variável teamFull, utilizada no ciclo anterior para verificar se o array de ids atingiu o número de jogadores pedidos (5), esteja com o valor verdadeiro apresentas os resultados caso contrário apresenta a mensagem “NOT ENOUGH PLAYERS TO MAKE IDEAL TEAM”.

Conclusão

Após o desenvolvimento do projeto concluímos que a linguagem C embora potente e muito versátil implica a criação de muito mais código para lidar com situações que em outras linguagens é mais simples principalmente no aspeto da gestão de memória que fica totalmente ao controlo do programador, influenciando assim a performance do programa final.

A criação e o planeamento das estruturas é essencial para a simplificação e estruturação do código, isto é visível especialmente neste projeto pelo facto de que é necessário carregar as informações dos ficheiros para o programa em desenvolvimento.

Houve algumas limitações durante o desenvolvimento do projeto, nomeadamente em relação à duplicação de código uma vez que não existe a possibilidade de criar funções genéricas como em JAVA, apesar de ser possível conseguir o mesmo efeito em C não foi implementado neste projeto, outras dificuldades envolveram gestão de memória, coisa que em outras linguagens não é um problema, e finalmente a relação de informação entre os dois ficheiros que em certos comandos levava a um elevado número de ciclos para obter a informação desejada entre os dois.