Ficheiro: fixsizedtypes.h
Este ficheiro define um conjunto de tipos e macros base. É definido um tipo que representará um byte sem sinal (uint8_t), bem como um tipo de dados que deverá representar uma expressão lógica (bool) e dois possíveis valores para o mesmo: true ou false.

Ficheiros: list.h e list.c

Podemos considerar que o coração do programa está nestes ficheiros. Eles definem uma lista genérica duplamente ligada com cabeçalho e rodapé[1], bem como um conjunto de funções para a manipular. O tipo de dados definido é:

```
typedef struct _list_t {
    void* data;
    struct _list_t* next;
    struct _list_t* prev;
} list_t;
```

Assumindo-se que em "data" se vai guardar sempre um ponteiro. Adicionalmente, também é definido um tipo compareFunc, que é um ponteiro para uma função, cujo objectivo será detalhado brevemente:

```
typedef int (*compareFunc)(void*, void*);
```

Para manipular este tipo de dados, são fornecidas as seguinte as seguintes funções:

```
list_t* ListNew(void);
```

Esta função cria uma lista e retorna um ponteiro para a mesma. Internamente, recorre à função malloc para alocar espaço para o cabeçalho e o rodapé, inicializando-os de forma apropriada (data=NULL para ambos os elementos).

```
list_t* ListAdd(list_t* list, void* g, compareFunc funct);
```

Esta função adiciona um elemento (g) de forma ordenada à lista list, recorrendo à função funct para a ordenação. Esta função receberá dois elementos que deverá comparar, retornando -1, 0 ou 1 se o primeiro elemento for, respectivamente, menor, igual ou maior do que o segundo. Podem ser utilizadas diferentes funções em diferentes alturas, consoante o critério que se queira utilizar. É retornado um ponteiro para o recém adicionado nó. Internamente, esta função recorre à função ListSearch, em seguida documentada. A memória em g não deverá ser libertada, pois a lista armazenará um ponteiro para ela.

```
list_t* ListSearch(list_t* list, void* key, compareFunc funct);
```

A função ListSearch recebe um elemento, key, e a função funct, já mencionada anteriormente, retornando o primeiro nó que seja "maior ou igual" (de acordo com os critérios de funct) a key. Se nenhum elemento nestas condições for encontrado, NULL é retornado.

```
void ListDel(list_t* l);
```

1 Definimos o nó de cabeçalho e o nó de rodapé como dois nós adicionais da lista-ligada, de conteúdo desprezável, a serem colocados no princípio e no fim, respectivamente, da lista duplamente ligada, por forma a permitir algoritmos mais genéricos e/ou mais eficientes em termos de tempo de processamento.

Esta função elimina o elemento l da lista ligada e liberta a memória associada a data, bem como a memória associada ao próprio nó.

```
void ListDelete(list_t* list);
```

A função ListDelete (não confundir com ListDel) elimina toda a lista e, por isso, deve receber um ponteiro para o cabeçalho. Internamente, o seu funcionamento é equivalente a invocar sucessicamente ListDel nos nós da lista, excepto no cabeçalho, eliminados à parte.

```
list_t* ListIterateNext(list_t* list);
list_t* ListIteratePrev(list_t* list);
```

As funções ListIterateNext e ListIteratePrev iteram sobre a lista, retornando um ponteiro para o próximo elemento ou para o elemento anterior, respectivamente.

```
bool ListIsHeader(list_t* list);
bool ListIsFooter(list_t* list);
```

Estas funções retornam verdadeiro se o nó list for, respectivamente, o cabeçalho ou o rodapé da lista. Retornam falso caso contrário.

```
list_t* ListFindNode(list_t* list, void* data);
```

Esta função retorna o nó cujo conteúdo seja igual (é utilizada aritmética de ponteiros) a data.

```
void ListDeleteNoFreeData(list_t* list);
```

A função ListDeleteNoFreeData é equivalente à função ListDelete mas não liberta associada aos dados de cada nó da lista (isto é: apenas liberta os nós). É particularmente útil quando se deseja criar uma lista que contém ponteiros para outras regiões que devem existir para além do tempo de vida da lista em questão.

É definida a seguinte macro auxiliar:

```
#define LIST_DATA(list, type) ((type*)list->data)
```

A macro tem como objectivo obter os dados de um dado nó da lista, fazendo a cast para o tipo apropriado. Por exemplo, para uma lista que guarde inteiros (ou seja, ponteiros para inteiros), poder-se-á fazer LIST_DATA(list, int).

Ficheiro GameList.h

Neste ficheiro é definida a estrutura Game e uma lista de ponteiros para jogos ordenados por data, que recorre a list_t. São também incluídas algumas funções para manipular esta estrutura.

No ficheiro datatypes.h existe a seguinte forward-declaration:

```
typedef struct _Game         Game;
```

Que, no ficheiro GameList.h, é concretizada:

```
struct _Game {
    uint8_t homePoints;
    uint8_t awayPoints;
    Date date;
    Team* homeTeam;
    Team* awayTeam;
};
```

Esta definição corresponde a um jogo. Deste modo, são armazenados os pontos da equipa da casa (homePoints) e da equipa visitante (awayPoints), bem como ponteiros para as equipas (homeTeam e awayTeam) e a data do jogo (date). A estrutura equipas será brevemente concretizada.

Estão presentes as seguintes declarações no ficheiro:

```
extern list_t*  Games;        /* Has all games ordered by date */
extern Game NULL_GAME;
```

A primeira define uma variável global, Games, que deverá conter uma lista com os jogos ordenados por data. A segunda define um jogo nulo. Estas variável é inicializada em globals.c como:

```
Game NULL_GAME = {
    0, 0,    // outcome
    {0,0,0}, // date
    NULL,    // home team
    NULL     // home team
};
```

É definida também a função:

```
list_t* GameListAddGame(list_t* list, Game g, bool update);
```

Que recebe um jogo, g, bem como uma lista (em princípio será Games), onde deverá inserir, ordenadamente, o referido jogo. O parâmetro update define se se deve actualizar o ficheiro onde estão armazenados os jogos. Também actualiza a lista de equipas ordenadas por classificação (ScoreboardList), bem como a lista de jogos de cada equipa (e respectiva pontuação) se esta já tiver sido criada. Para isto, recorre às funções TeamUpdateGameListCache e ScoreboardListUpdate. Para a actualização do ficheiro de dados, utiliza a função UpdateGames.
Uma vez que esta função recorre à já documentada ListAdd, precisa de lhe fornecer uma função que compare duas equipas e retorne a diferençe em termos de “datas” dela. Para isso, é definida e utilizada a função:

```
int compareDatesAux(void* d1, void* d2);
```

Que simplesmente invoca a função compareDates após aplicar uma cast e o operador de desreferência em d1 e d2.

Para eliminar a lista de jogos, pode-se recorrer à já documentada função ListDelete.

Por fim, são definidas as seguintes macros:

```
#define OUTCOME_HOMEWIN 1
#define OUTCOME_DRAW    2
#define OUTCOME_AWAYWIN 3
```

Que são uma maneira rápida, em conjunto com a função getOucomeFromGame, de obter o vencedor de uma partida. Esta função recorre a uma comparação dos pontos obtidos por cada equipa.

Ficheiros: Team.c e Team.h

Em Team.h é definida a estrutura Team, que representa uma equipa. Em datatypes.h, está presente a seguinte forward-declaration:

```
typedef struct _Team         Team;
```

Bem como a macro:

```
#define NAME_SIZE 31
```

No ficheiro Team.h há uma concretização:

```
struct _Team {
    char name[NAME_SIZE];
    char location[NAME_SIZE];
    int points; // -1 indicates not cached yet
    list_t* gameList;
};
```

A definição de estrutura Team assum que cada equipa tem um nome e uma localidade, com tamanho máximo de 30 caracteres (mais o terminador nulo). Também se recorre a um sistema de cache, mantendo-se os pontos de cada equipa e uma lista (definida em TeamGameList) com ponteiros para os nós da lista de jogos, no que diz respeito aos jogos em que essa equipa participa, ordenados por data. Estes dois campos apenas serão iniciados/utilizados quando necessário. Por exemplo, se for necessário ordenar as equipas por ordem de pontos, então tanto gameList (que estava a NULL) como points (que estava a -1) serão iniciados. A partir do momento em que points e gameList são iniciados, serão sempre mantidos actualizados.

As equipas serão mantidas num array, variável global, Teams, alocado dinamicamente.

Para manipular as equipas, são definidas várias funções:

```
void TeamPointsUpdateWithGame(Team* team, Game* game, bool remove);
```

Esta função actualiza o campo points da equipa team com o jogo em game[2]. Se remove for verdadeiro, então os pontos são subtraídos e não adicionados (este campo existe para ser utilizado aquando da remoção de um jogo). Internamente recorre à já documentada função getOutcomeFromGame.

```
void TeamUpdateGameListCache(Team* team, list_t* gameNode);
```

Esta função tem como finalidade actualizar a lista de jogos da equipa team com o nó da lista de jogos (definida em GameList.h), isto é, adicioná-lo a team->gameList e, se necessário, actualizar os pontos da equipa (recorrendo a TeamPointsUpdateWithGame). Se gameNode for NULL, então a função deverá popular team->gameList com todos os jogos dessa equipa se tal ainda não tiver sido feito. Destaque para o facto de esta função não alterar team->points, excepto quando este campo já foi iniciado (pela função TeamGetPoints). Internamente, recorre à função TeamGameListAppend, que será em breve documentada.

2 De acordo com o critério: 3 pontos para a vitória, 1 para o empate e 0 para a derrota.

```
int TeamGetPoints(Team* team);
```

Esta função retorna os pontos da equipa. Se necessário, invoca TeamUpdateGameListCache para criar a lista de jogos e também inicia team->points. Para a contagem de pontos, quando a lista de jogos já foi criada mas os pontos ainda não foram contabilizados, itera sobre esta mesma lista e recorre a TeamPointsUpdateWithGame.

```
void TeamDelGame(Team* team, Game* g);
```

A função TeamDelGame remove o nó da lista team->gameList que contém o ponteiro para o nó cujo jogo é g. Na prática, isto significa que a informação (incluindo os pontos) relativa ao jogo g é totalmente eliminada desta equipa, como se ele não tivesse acontecido. Internamente, recorre à função TeamGameListDelGame, que será em breve documentada.

```
Team* TeamFind(const char* _name);
```

TeamFind é uma função que tem por objectivo encontrar a equipa cujo nome é _name (case insensitive) e retornar um ponteiro para esta. Se tal equipa não for encontrada, NULL é retornado. A pesquisa é feita utilizando um algoritmo “binary search”.

Também é declarada uma variável global (definida em globals.c):

```
extern list_t*  LastGameList; /* Points to the last listed game list.*/
```

Que aponta para a última lista de jogos de cada equipa apresentada no ecrã ou NULL, caso o utilizador tenha visto todos os jogos sem estarem associados a nenhuma equipa em particular. Esta variável é utilizada por se utilizarem “ids” como as identificações dos jogos. O id corresponde ao nº do nó de LastGameList (a começar em 0) ou de Games. Por exemplo, se o utilizador quiser apagar o primeiro jogo da equipa “FC Porto” depois de ter visto os seus jogos, pode apenas escrever “apagar 0”.

Ficheiros: TeamGameList.h e TeamGameList.c

Já foi referida a existência de uma lista com ponteiros para nós da lista com ponteiros para jogos. Funções auxiliares para manipular esta lista são mantidas em TeamGameList.h e TeamGameList.c que, evidentemente, assentam em list_t.

```
list_t* TeamGameListAdd(list_t* list, list_t* g);
```

Esta função é responsável por adicionar, por ordem de data, o elemento g da lista definida em GameList à lista (TeamGameList) list. Para a ordenação, recorre à função compareDatesFromNodeAux implementada como:

```
int compareDatesFromNodeAux(void* d1, void* d2) {
    return compareDatesAux(((Game*)((list_t*)d1)->data),
                           ((Game*)((list_t*)d2)->data));
}
```

O que quer dizer que delega a tarefa de comparar datas para a já documentada função compareDatesAux que, por sua vez, recorre a compareDates.

```
list_t* TeamGameListAppend(list_t* list, list_t* g);
```

A função TeamGameListAppend insere o ponteiro g para o nó da lista definida em GameList no fim da lisa list. Foi criada por questões de eficiência aquando do seu uso em TeamUpdateGameListCache.

```
list_t* TeamGameListDel(list_t* l);
```

Esta função remove o elemento l da lista a que este pertence, sem libertar a memória associada a l->data. É utilizada para remover elementos de uma TeamGameList sem os apagar da lista original (GameList) a que pertencem.

```
void TeamGameListDelGame(list_t* list, Game* g);
```

A função TeamGameListDelGame encontra o nó correspondente ao jogo g e invoca TeamGameListDel nesse nó. Para procurar elementos, recorre à função compareGamesAux, implementada como:

```
int compareGamesAux(list_t* d1, Game* g) {
    return (GAMELIST_GAME(d1) == g);
}
```