Mini Projeto
Princípios de Programação Procedimental

João Ricardo Maximiano Leitão Ribeiro Lourenço	**2011151194**
João Afonso Libório Cardoso	**2011151968**

Índice

FixedSizedTypes.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.

String.h e String.c

Funções

Neste ficheiros são definidas várias funções auxiliares para a manipulação de strings e ficheiros, que não se encontram disponíveis nas bibliotecas padrão. É de destacar:

```
size_t readString(char* str, size_t maxlen)
size_t readStringFile(char* str, size_t maxlen, FILE* f)
```

Lêem até "maxlen" caracteres (incluindo o terminador) da entrada padrão ou de um dado ficheiro, respectivamente. Os caracteres são copiados para str. Devolve o tamanho da string ou 0, em caso de erro.

```
bool strToResult(char* str, uint8_t* pts1, uint8_t* pts2)
```

Verifica se str segue o formato "%d-%d". Caso siga, copia cada um dos números encontrados para pts1 e pts2. Devolve true em caso de sucesso.

```
bool isStrNumber(const char* s)
```

Devolve verdadeiro caso o conteúdo da string "s" seja um número natural.

```
void toLower(char s[])
```

Converte os caracteres da string providenciada para letras minúsculas.

Date.h e Date.c

Tipos

Neste ficheiros é definido o tipo "Date", que representa um dia de um dado ano, bem como funções destinadas à sua manipulação.

```
typedef struct _Date {
    uint8_t day, month;
    unsigned int year;
} Date;
```

E são declaradas duas constantes que definem o valor máximo e mínimo deste tipo

```
Date DATEMAX = {(uint8_t)-1, (uint8_t)-1, (unsigned int)-1};
Date DATEMIN = {0,0,0};
```

Funções

```
int compareDates(Date d1, Date d2)
```

Determina qual das duas datas é anterior à outra. Devolve 0 caso sejam iguais, -1 caso “d1” seja anterior, ou 1 caso “d1” seja posterior.

```
bool isValidDate(Date d)
```

Devolve verdadeiro caso a data providenciada for válida no calendário gregoriano (ex: o mês está entre 1 e 12).

```
void printDate(Date d, bool longFormat)
```

Imprime a data devidamente formatada para a saída padrão.

```
Date getDateFromUser(const char* msg)
```

Pede ao utilizador que submeta uma data na entrada padrão (em qualquer variação dos formatos “d/m/yy” e “dd/mm/yyyy”). A string “msg” é impressa na saída padrão, para informar do pedido o utilizador.

List_t.h e List_t.c

Tipos

Podemos considerar que o coração do programa está nestes ficheiros. Eles definem uma lista genérica duplamente ligada com cabeçalho e rodapé, 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*);
```

Funções

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

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);
```

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);
```

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);
```

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);
```

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);
```

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);
```

Devolvem 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);
```

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

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

Esta função 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).

GameList.h

Tipos

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.

Declarações

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
};
```

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.

Funções

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

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 diferença 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.

Team.c e Team.h

Tipos

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 assume 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.

Funções

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

Actualiza o campo points da equipa team com o jogo em game. 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.

Os pontos das equipas são determinados de acordo com o critério: 3 pontos para a vitória, 1 para o empate e 0 para a derrota.

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

Actualiza a lista de jogos da equipa team com o nó da lista de jogos (definida em GameList.h), isto é, adiciona-o a team->gameList e, se necessário, actualiza 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.

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

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);
```

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);
```

Encontra a equipa cujo nome é name (case insensitive) e retorna um ponteiro para esta. Se tal equipa não for encontrada, NULL é retornado. A pesquisa é feita utilizando um algoritmo "binary search".

Declarações

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

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".

TeamGameList.h e TeamGameList.c

Funções

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);
```

Adiciona, 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);
```

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

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

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);
```

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);
}
```

ScoreBoardList.h e ScoreBoardList.c

Funções

Para apresentar ao utilizador as equipas ordenadas por pontos, foi criada uma lista (do tipo list_t), cujo conteúdo serão ponteiros para equipas (Team*). Estes ficheiros definem um conjunto de funções para manipular essa estrutura.

```
list_t* ScoreboardListUpdate(list_t* list, Game* game)
```

A função ScoreboardListUpdate é muito análoga à função TeamUpdateGameListCache. Se a lista list ainda não tiver sido criada, então é criada e todas as equipas são colocadas, por ordem de pontos, nesta lista, recorrendo à função compareTeamsByPointsAux, brevemente documentada. Se, por outro lado, a lista já tiver sido criada, a função investiga quais as equipas que participam no jogo game, remove-as da lista e volta a adicioná-las. Para que este processo funcione é necessário que a função apenas seja chamada depois de os pontos de cada equipa já terem sido alterados em função da adição ou remoção do jogo game. Para localizar a equipa na lista, recorre-se à já documentada função ListSearch, passando-lhe como função de ordenação a seguinte função:

```
int compareTeamsAux(void* d1, void* d2) {
  return (d1==d2) ? 0 : -1;
}
```

A função compareTeamsByPointsAux, já referida, está implemantada da seguinte forma:

```
int compareTeamsByPointsAux(void* d1, void* d2) {
  int pts1, pts2;
  pts1 = TeamGetPoints((Team*)d1);
  pts2 = TeamGetPoints((Team*)d2);
  return pts2-pts1;
}
```

Por último:

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

Tem uma implementação e funcionalidade igual aTeamGameListDel.

Saveload.c e Saveload.h

Formato de Ficheiros

Os dados de equipas e de jogos têm de ser armazenados e lidos usando ficheiros. Assim, as equipas são armazenadas em “team.txt”, e os jogos em “games.txt”.

“team.txt” segue os seguinte formato:

```
Nome: <nome da equipa>
Localidade: <localidade da equipa>

Nome: <outra equipa>
Localidade: <outra localidade>
```

É admitido um número de espaços variável entre todos os elementos. Para separar as equipas, um número variável de quebras de linha pode ser inserido.

"games.txt" segue o seguinte formato:

```
Casa: <nome da equipa da casa>
Fora: <nome da equipa de fora>
Data: <data, em qualquer variação dos formatos "d/m/yy" e "dd/mm/yyyy">
Resultado: <pontos de casa e de fora, no formato "ccc-fff">
```

Em que a separação dos elementos segue as mesmas regras que "team.txt". Relativamente à data, caso sejam inseridos dois ou um algarismos para o ano, considera-se que é relativamente ao ano 2000 (ex: "9" é "2009").

Se a leitura e interpretação de alguma linha falhar, o utilizador será informado do erro, bem como da linha em questão.

Funções

```
void ReadTeams()
```

Aloca a memória para a variável global "Teams" e insere, por ordem alfabética, as equipas no array.

```
void ReadGames()
```

Constrói e aloca a memória necessária para os elementos da lista ligada "Games".

```
void WriteGames()
```

Rescreve o conteúdo da lista ligada "Games" no ficheiro "games.txt". Necessário para guardar alterações feitas aos jogos.

Declarações

Todas as funções recorrem a uma macro para informar o utilizador de erros:

```
#define ERROR_OUT(msg, ...) do {printf("ERROR: " msg "\n" , __VA_ARGS__); fclose(file); exit(-1);}
while(0)
```

Que, por questões de portabilidade, deverá sempre receber mais do que um argumento, mesmo que a mensagem aparentemente não o exija.

Actions.h e Actions.c

Funções

Neste ficheiro são definidas funções auxiliares responsáveis por agir sobre o dados do programa, a pedido do utilizador.

```
void clearScreen(void)
```

Define uma maneira portável de limpar o ecrã.

```
Team* getTeamFromInput(char* input)
```

Devolve um ponteiro para a equipa referenciada em input. Input pode conter o nome da equipa, case-insensitive e com um número variável de espaços à sua volta.

Alternativamente, se for um número será considerado o índice da equipa (1ª equipa terá índice 0, 2ª equipa, índice 1.

```
void printGame(size_t id, Game* game);
```

Imprime uma linha com os dados de um jogo (id, equipas, pontos e data) formatados numa tabela. Assume que existe um terminal com pelo menos 91 caracteres. O id e a sua funcionalidade já foram antes documentados.

```
void TablePrintTeam(size_t i, bool showPos, Team* t)
```

Análoga à função "printGame", mas para equipas. Caso showPos seja verdadeiro, a posição da equipa (i) também será impressa.

```
void TeamNameError(char str[])
```

Notificar o utilizador de um erro a inserir o nome de uma equipa.

```
void RequestPoints(char msg[], uint8_t* points);
```

Imprime a string "msg" e pede pontos ao utilizador, garantindo que são válidos (têm de estar entre 0 e 255 para evitar overflow). A função apenas termina quando o utilizador tiver inserido dados válidos.

```
void NewGame(void)
```

Adiciona, interactivamente, um novo jogo à lista de jogos, adicionando-o à lista de jogos de cada equipa, se necessário, actualizando, também se necessário, ScoreboardList, bem como o ficheiro "games.txt".
Apenas termina quando o utilizador tiver inserido dados válidos.

```
void DeleteGame(char* input);
```

Quando o utilizador escreve "apagar", a função DeleteGame é chamada com este comando e com o input que se seguiu a "apagar". Se, de facto, o utilizador tiver escrito algo para além de "apagar", então assume-se que é o id de um jogo. Se o utilizador não tiver inserido input adicional, é aprsentada no ecrã a mais recente lista de jogos e pergunta-se ao utilizador se quer apagar um jogo desta ou de outra lista de jogos, até que este tenha de inserir um id. Nessa altura, o jogo será removido de todos os locais possíveis (listas de cache), com os pontos de cada equipa actualizados (apenas se necessário), bem como ScoreBoardList.

main.c

Funções

O ficheiro main.c define apenas duas funções.

```
void cleanup(void)
```

Responsável libertar a memória associada a todas as estruturas, quer em caso de execução correcta do programa, quer em caso de um fim inesperado (por exemplo, durante a leitura dos dados).

```
int main()
```

Responsável pela maior parte da interactividade do programa, funcionando como um interpretador de comandos, que podem ser vistos com o comando “help”.