Instituto Superior Técnico

Engenharia Eletrotécnica e de Computadores

Programação

Blackjack

Projecto final de Programação

Docente: João Ascenso

Autores:

João Pinheiro	84086	joao.castro.pinheiro@tecnico.ulisboa.pt
João Freitas	84093	joao.m.freitas@tecnico.ulisboa.pt

Grupo 25

2015/2016 - Primeiro Ano - Segundo Semestre
Maio de 2016

```c
/* O sistema de ajuste de apostas utilizado neste projecto segue as instruçes
 * explicadas aqui: http://casinogambling.about.com/od/blackjack/a/hilo.htm
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
#include "error.h"
#include "logic.h"
#include "ea.h"

/*
 * O formato do ficheiro de estratégia das EAs é:
 * 10 (HARD_LINES) linhas com 10 caracteres cada para as decises Hard
 * uma linha em branco (um \n)
 * 8 (SOFT_LINES) linhas com 10 caracteres cada para as decises Soft
 * os caracteres podem ser:
 * H - hit
 * S - stand
 * R - surrender
 * D - double senão hit
 * E - double senão stand
 */

void write_matrix(Move ***matrix, FILE *file, int lines)
{
    char buffer[COLUMNS+2] = {0}; // COLUMNS + '\n' e '\0'
   *matrix = (Move **) ecalloc(lines, sizeof(Move *));

    for (int i = 0; i < lines; i++) {
      fgets(buffer, COLUMNS+2, file);
      // Se o caracter exatamente aps COLUMNS colunas da linha não for \n,
      // sabemos que a linha não tem exatamente COLUMNS caracteres.
      if (buffer[COLUMNS] != '\n') {
         fprintf(stderr, "Erro: Ficheiro de estratégia das EAs mal formatado.\n");
         exit(EXIT_FAILURE);
      }
        (*matrix)[i] = (Move *) ecalloc(10, sizeof(Move));
         for (int j = 0; j < COLUMNS; j++)
            (*matrix)[i][j] = buffer[j];
    }
}

void destroy_matrix(Move **matrix, int lines)
{
    for (int i = 0; i < lines; i++)
      free(matrix[i]);

   free(matrix);
}


// Ler a estrategia do ficheiro de configuração
Strategy *read_strategy(char *filename)
{
   char check[2] = {0}; // \n e \0
    FILE *config_file = efopen(filename, "r");

    Strategy *strategy = (Strategy *) ecalloc(1, sizeof(Strategy));

    strategy->hard = NULL;
    strategy->soft = NULL;
```

```
    write_matrix(&strategy->hard, config_file, HARD_LINES);

    // Verificar \n de separação das matrizes
    fgets(check, 2, config_file);
    if (check[0] != '\n') {
      fprintf(stderr, "Erro: Ficheiro de estratégia das EAs mal formatado.\n");
      exit(EXIT_FAILURE);
   }

    write_matrix(&strategy->soft, config_file, SOFT_LINES);

   fclose(config_file);

    return strategy;
}

/*
 * Decide que matriz utilizar: hard ou soft
 * Calcula a coluna e linha da matriz
 * Retorna a decisão a tomar
 */
Move get_decision(Player *player, Card *house_card, Strategy *strategy)
{
    bool ace = false;
    int line = 0, column = 0;
    Stack *aux = player->cards;

    //Verificar se a ases
    while(aux) {
        if (aux->card->id == 12)
            ace = true;
        aux = aux->next;
    }

    // Calcular a coluna da matriz
    if (house_card->id > 0 && house_card->id < 8)
        column = house_card->id;
    else if (house_card->id >=8 && house_card->id < 12)
        column = 8;
    else if (house_card->id == 12)
        column = 9;

    // Calcular a linha da matriz soft
    if (ace) {
      // Dois ases correspondem  primeira linha da matriz soft
      if (player->points == 12)
         line = 0;
        else if (player->points > 12 && player->points < 19)
            line = player->points - 12;
        else if (player->points >= 19)
            line = 7;

        return strategy->soft[line][column];
    }
    // Calcular a coluna da matriz hard
    else {
        if (player->points >= 4 && player->points <= 8)
            line = 0;
        else if (player->points > 8 && player->points < 17)
            line = player->points - 8;
        else if (player->points >= 17)
            line = 9;

        return strategy->hard[line][column];
```

```
    }
}


/*
 * Encontra proximo jogador
 * Usa o valor de retorno de get_decision para escolher a proxima ação
 */
void ea_make_decision(List *players, Player *house, Megadeck *megadeck, Strategy *strategy)
{
   bool can_double = false;
   List *aux = find_active_player(players);
   Player *cur_player = (Player *) aux->payload;

   Card *house_card = house->cards->next->card;
   Move decision = get_decision(cur_player, house_card, strategy);

   switch (decision) {
      case H:
         player_hit(players, house, megadeck);
         break;

      case S:
         stand(players, house, megadeck);
         break;

      case R:
         surrender(players, house, megadeck);
         break;


      case D:
         can_double = double_bet(players, house, megadeck);
         if (!can_double) {
            player_hit(players, house, megadeck);
         }
         break;
      case E:
         can_double = double_bet(players, house, megadeck);
         if (!can_double) {
            stand(players, house, megadeck);
         }
         break;

      default:
         // Isto nunca deverá acontecer
         fprintf(stderr, "Erro: Decisão de EA inesperada.\n");
         exit(EXIT_FAILURE);
         break;
   }
}


/*
 * Cada jogador começa com a contagem = 0
 * Em cada ronda são contadas as cartas e no fim da ronda ao count dos jogadores
 * é somada a contagem da ronda
 */

//Conta cartas segundo a estrategia hi-lo
void count_cards(Card *new_card, Megadeck *megadeck)
{
   if (new_card->id < 5)
      megadeck->round_count++;
   else if (new_card->id > 7)
```

```
     megadeck->round_count--;
}

// soma player->count com megadeck->round_count
void update_count(List *players, Megadeck *megadeck)
{
   List *aux = players->next;
   Player *cur_player = NULL;

   while (aux) {
      cur_player = (Player *) aux->payload;
      if (cur_player->type == EA)
         cur_player->count += megadeck->round_count;

      aux = aux->next;
   }

   megadeck->round_count = 0;
}

/* Altera a bet do jogador
 * A aposta original do jogador éuma unidade
 * true_count = contagem / numero de baralhos
 * A nova aposta no jogador éigual a 2*true_count unidades
 * Se true_count <= 0 a nova aposta e igual a 1 unidades
 */
void hi_lo(Player *player, Megadeck *megadeck)
{
    double new_bet = 0;
   double decks_left = round(((double) megadeck->cards_left)/DECK_SIZE + 1);
   int true_count = round(player->count/decks_left);

    if (true_count <= 0)
      new_bet = player->orig_bet;
    else
        new_bet = 2 * true_count * player->orig_bet;


    if (player->type == EA) {
      if (player->money > new_bet)
         player->bet = new_bet;
      else
         player->bet = player->money;
   }
}
```

```
#ifndef EA_H
#define EA_H

#include <stdbool.h>
#include "logic.h"

#define COLUMNS 10
#define HARD_LINES 10
#define SOFT_LINES 8
#define HI_LO

void write_matrix(Move ***matrix, FILE *file, int lines);
void destroy_matrix(Move **matrix, int lines);

Strategy *read_strategy(char *filename);
Move get_decision(Player *player, Card *house_card, Strategy *strategy);
void ea_make_decision(List *players, Player *house, Megadeck *megadeck, Strategy *strategy);
```

```
void count_cards(Card *new_card, Megadeck *megadeck);
void update_count(List *players, Megadeck *megadeck);
void hi_lo(Player *player, Megadeck *megadeck);

#endif
```

```
#include <stdlib.h>
#include <stdio.h>

void *ecalloc(size_t nmemb, size_t size)
{
   void *memory = calloc(nmemb, size);

   if (memory == NULL) {
      fprintf(stderr, "Erro: Impossvel alocar memria.\n");
      exit(EXIT_FAILURE);
   }

   return memory;
}

FILE *efopen(const char *path, const char *mode)
{
   FILE *file = fopen(path, mode);

   if (file == NULL) {
      fprintf(stderr, "Erro: Impossvel abrir ficheiro %s.\n", path);
      exit(EXIT_FAILURE);
   }

   return file;
}
```

```
#include <stdio.h>

void *ecalloc(size_t nmemb, size_t size);
FILE *efopen(const char *path, const char *mode);
```

```
/*
 * Leitura e escrita de ficheiros
 * Lê o ficheiro de configuração dos jogadores
 * Escreve o ficheiro de estatisticas
 * Lê a os parametros de um jogador quando énecessario inserir um jogador
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "file.h"
#include "logic.h"
#include "error.h"

/* Lê o ficheiro de configuração dos jogadores
 * Numero de baralhos e jogadores
 */
Config *read_config(char *filename)
{
   char buffer[MAX_LINE_LEN];

   Config *config = NULL;
```

```
   config = (Config *) ecalloc((size_t) 1, sizeof(Config));

   FILE *config_file = efopen(filename, "r");

    //Parametros gerais de configuração:
    //Numero de jogadores e numero de baralhos
   fgets(buffer, MAX_LINE_LEN, config_file);
   sscanf(buffer, "%d-%d", &(config->num_decks), &(config->num_players));

   if (config->num_decks > 8 || config->num_decks < 4){
      fprintf(stderr, "Erro: nmero de baralhos invalido.\n");
      exit(EXIT_FAILURE);
   }

   if (config->num_players > 4 || config->num_players < 1) {
      fprintf(stderr, "Erro: nmero de jogadores invalido.\n");
      exit(EXIT_FAILURE);
   }
    //Leitura dos paramtros de configuracão de cada jogador
   for (int i=0; fgets(buffer, MAX_LINE_LEN, config_file) != NULL && i < config->num_players; i++)
      config = read_player(buffer, config, i);

   fclose(config_file);

   return config;
}

/*
 * Leitura dos parametros de configuração de cada jogador
 * Recebe uma string e separa os parametros de configuração com strtok
 */
Config *read_player(char *line, Config *config, int count)
{
    // strtok separa o buffer no caracter '-'
   char *str = strtok(line, "-");

    //Leitura do tipo do jogador
   if (strcmp(str, "HU") == 0)
      config->player_type[count] = HU;
   else if (strcmp(str, "EA") == 0)
      config->player_type[count] = EA;
   else {
      fprintf(stderr, "Erro: tipo de jogador inválido.\n");
      exit(EXIT_FAILURE);
   }

   str = strtok(NULL, "-");
   if (strlen(str) > MAX_PLAYER_NAME) {
      fprintf(stderr, "Erro: nome do jogador demasiado grande (Máx. 8 caracteres).\n");
      exit(EXIT_FAILURE);
   }
   strcpy(config->player_names[count], str);

    //Ultimo segmento da string
    str = strtok(NULL, "\0");
   sscanf(str, "%d-%d", &config->money[count], &config->bets[count]);
   if (config->money[count] < 10 || config->money[count] > 500) {
      fprintf(stderr, "Erro: valor inicial de dinheiro inválido.\n");
      exit(EXIT_FAILURE);
   }

   if (config->bets[count] < 2 ||
      config->bets[count] > config->money[count] / 4) {
      fprintf(stderr, "Erro: valor da aposta invalido!\n");
      exit(EXIT_FAILURE);
```

```
   }

   return config;
}

/* Vai buscar uma linha a stdin.
 * Modifica o buffer por referência.
 * O buffer fica vazio se o fgets() der overflow ou se a input for vazia.
 * Senão, o buffer fica com a string de input, sem o \n.
 */
void get_line(char buffer[MAX_PLAYER_NAME+2])
{
   int newline = 0;
   int c = 0;
   fgets(buffer, MAX_PLAYER_NAME+2, stdin);

   // localização do \n
   newline = (int) strcspn(buffer, "\n");

   // se não existir (ou seja, newline éo comprimento da string inserida),
   // sabemos que a string de stdin émaior que o buffer pode conter.
   if (newline == MAX_PLAYER_NAME+1) {
      strcpy(buffer, "");
      // Consumir o resto do buffer de stdin
      while ((c = getchar()) != '\n' && c != EOF);
      return;
   }
   // se existir, substituir por \0.
   // neste caso se buffer estiver vazio, permanece vazio.
   else
      buffer[newline] = '\0';
}

//  Ler novo valor da aposta a partir stdin
void get_new_bet(List *players)
{
   char buffer[MAX_PLAYER_NAME+2] = {0}; // newline + nullbyte
   bool correct = false;
   List *aux = players->next;
   Player *cur_player = NULL;

   printf("Insira o nome do jogador a modificar a aposta: ");
   get_line(buffer);

   if (buffer[0] == '\0') {
      puts("Jogador não encontrado. Tente novamente primindo a tecla <b>.");
      return;
   }

   correct = false;
    //Verificar se o jogador existe
   while (aux && !correct) {
      cur_player = (Player *) aux->payload;
      if (strcmp(buffer, cur_player->name) == 0 && !correct)
         correct = true;
      else
         aux = aux->next;
   }

   if (!aux) {
      puts("Jogador não encontrado. Tente novamente primindo a tecla <b>.");
      return;
   }

   correct = false;
```

```
   cur_player = (Player *) aux->payload;
   long new_bet = 0;
   do {
      printf("Insira o novo valor da aposta do jogador %s: ", cur_player->name);
      get_line(buffer);

      if (buffer[0] == '\0')
         puts("Nova aposta inválida.");
      else {
         new_bet = strtol(buffer, NULL, 10);
         // o dinheiro do jogador está (essencialmente) garantido
         // de estar abaixo de INT_MAX (a não ser que se jogue mesmo muito)
         // fazendo com bet pertencente a [1, money]
         if (new_bet > cur_player->money || new_bet < 1)
            printf("Nova aposta inválida [1-%d].\n", cur_player->money);
         else
            correct = true;
      }
   } while (!correct);

   cur_player->bet = (int) new_bet;
}



/*
 * Obter o valor do dinheiro, tipo, nome e aposta do jogador
 * Pede ate obter um valor correto
 */
Player *get_new_player(int pos)
{
   char buffer[MAX_PLAYER_NAME+2] = {0};
   bool correct = false;
   Type type = HU;
   char name[MAX_PLAYER_NAME+1] = {0};
   int money = 0;
   int bet = 0;
   long money_tmp = 0;
   long bet_tmp = 0;
   Player *new_player = NULL;

   printf("Escolheu o %d lugar.\n", pos);

    correct = false;
   do {
      printf("Introduza o tipo do jogador [HU ou EA]: ");
      get_line(buffer);

      if (buffer[0] == '\0')
         puts("Tipo de jogador inválido [HU ou EA].");
      else {
         if (strcmp(buffer, "HU") == 0) {
            type = HU;
            correct = true;
         }
         else if (strcmp(buffer, "EA") == 0) {
            type = EA;
            correct = true;
         }
         else
            puts("Tipo de jogador inválido (HU ou EA).");
      }
   } while (!correct);

   correct = false;
```

```
   do {
      printf("Introduza o nome do jogador [máx. 8 carac.]: ");
      get_line(buffer);

      if (buffer[0] == '\0')
         puts("Nome do jogador inválido. Este tem no máximo 8 caracteres.");
      else {
         strcpy(name, buffer);
         correct = true;
      }
   } while (!correct);

   correct = false;
   do {
      printf("Introduza o dinheiro do jogador: ");
      get_line(buffer);


      if (buffer[0] == '\0')
         puts("Dinheiro inválido.");
      else {
         money_tmp = strtol(buffer, NULL, 10);
         if (money_tmp <= 1 || money_tmp > INT_MAX)
            printf("Quantidade de dinheiro inválida [de 1 a %d].\n", INT_MAX);
         else {
            correct = true;
            money = (int) money_tmp;
         }
      }
   } while (!correct);


   correct = false;
   do {
      printf("Introduza a aposta do jogador: ");
      get_line(buffer);

      if (buffer[0] == '\0')
         puts("Aposta inválida.");
      else {
         bet_tmp = strtol(buffer, NULL, 10);
         if (bet_tmp > money_tmp || bet_tmp <= 0)
            printf("Aposta inválida [de 1 a %d].\n", money);
         else {
            correct = true;
            bet = (int) bet_tmp;
         }
      }
   } while (!correct);

    // Alocar espaço para o jogador e escrever a configuração
   new_player = (Player *) ecalloc((size_t) 1, sizeof(Player));

   new_player->ingame = true;
   new_player->type = type;
   strcpy(new_player->name, name);
   new_player->money = money;
   new_player->bet = bet;

   return new_player;
}


// Escrever o ficheiro de estatisticas
void write_stats(List *players, Player *house, List *old_players)
```

```
{
    FILE *stats = NULL;
    stats = efopen("stats.txt" , "w");

    fprintf(stats, "Jogador\t\tTipo\tJogos\tVitorias\tEmpates\tDerrotas\tDinheiro\n");

   write_stats_players(stats, players);
   write_stats_players(stats, old_players);

    // O dinheiro da casa esta em modulo. E indicado se a casa perdeu ou ganhou
    // dinheiro
   if (house->money < 0)
      fprintf(stats, "A casa perdeu: %d \n", -1*house->money);
   else if (house->money > 0)
      fprintf(stats, "A casa ganhou: %d \n", house->money);
   else if (house->money == 0)
      fprintf(stats, "A casa não ganhou nem perdeu dinheiro.\n");

   fclose(stats);
}

// Escrever as estatisticas dos jogadores
void write_stats_players(FILE *stats, List *players)
{
   List *aux = players->next;
   Player *cur_player = NULL;
    while (aux) {
        cur_player = (Player *) aux->payload;
        if (cur_player->type == VA) {
            aux = aux->next;
            continue;
        }
        fprintf(stats, "%s\t", cur_player->name);
        if (strlen(cur_player->name) < 8)
            fprintf(stats, "\t");
        if (cur_player->type == EA)
            fprintf(stats, "EA\t");
        else if (cur_player->type == HU)
            fprintf(stats, "HU\t");
        fprintf(stats, "%d\t", cur_player->wins+cur_player->losses+cur_player->ties);
        fprintf(stats, "%d\t\t", cur_player->wins);
        fprintf(stats, "%d\t", cur_player->ties);
        fprintf(stats, "%d\t\t", cur_player->losses);
        fprintf(stats, "%d  \n", cur_player->money);
        aux = aux->next;
    }
}
```

```
#ifndef FILE_H
#define FILE_H

#include "logic.h"
#define MAX_LINE_LEN 100

Config *read_player(char *line, Config *game_config, int count);
Config *read_config(char *filename);

void get_line(char buffer[MAX_PLAYER_NAME+2]);
void get_new_bet(List *players);

Player *get_new_player(int pos);

void write_stats(List *players, Player *house, List *old_players);
void write_stats_players(FILE *stats, List *players);
```

```
#endif
```

```
/*
 * Implementação das listas que vamos usar no projeto.
 * Estas são listas de payload genérico, doubly-linked com dummy head node.
 *
 * A desvantagem de utilizar listas de payload genérico éque
 * obrigamos o utilizador a fazer cast do payload para o que ele
 * precisar, o que pode ser chato, adicionar complexidade ao cdigo
 * e/ou criar bugs difceis de perceber se nos esquecermos de
 * fazer cast da payload... (ou seja, fazia-se dereference dum
 * void pointer, o que éilegal) _E_ o utilizador tem de fazer free()
 * da payload manualmente.
 *
 * Mas, por outro lado, o programa torna-se mais modular,
 * mais fácil de compreender e abstrai-se assim toda a parte
 * das listas.
 *
 */
#include <stdlib.h>
#include <stdio.h>
#include "list.h"
#include "error.h"


// Aceder a um no numa posilão especifica da lista
List *list_follow(List *head, int pos)
{
   if (pos < 0) {
      fprintf(stderr, "Erro: tentou-se aceder a um n não existente na lista.\n");
      exit(EXIT_FAILURE);
   }

   List *aux = head;
   for (int i = 0; i < pos; i++) {
      if (aux != NULL) {
         aux = aux->next;
      }
      else {
         fprintf(stderr, "Erro: tentou-se aceder a um n não existente na lista.\n");
         exit(EXIT_FAILURE);
      }
   }

   return aux;
}

// Inserir um no numa posição especifica da lista
void list_insert_pos(List *head, int pos, void *payload)
{
   List *aux = list_follow(head, pos - 1);

   List *new = (List *) ecalloc((size_t) 1, sizeof(List));
   new->payload = payload;

   new->next = aux->next;
   if (aux->next != NULL)
      aux->next->prev = new;
   else {
      // inserting at the tail, no need to set aux->next->prev
   }

   new->prev = aux;
```

```
   aux->next = new;
}


//Inserir na tail
void list_append(List *head, void *payload)
{
   List *aux = head;
   while (aux->next != NULL)
      aux = aux->next;
   List *tail = aux;

   List *new_tail = (List *) ecalloc((size_t) 1, sizeof(List));

   new_tail->payload = payload;

   new_tail->next = NULL;
   new_tail->prev = tail;
   tail->next = new_tail;
}

//Remover um no da lista
void *list_remove(List *node)
{
   List *to_rm = node;
   void *payload = to_rm->payload;

   if (node->next != NULL) {
      node->next->prev = to_rm->prev;
   }
   else {
      // removing tail, it has no next, skip.
   }

   if (node->prev != NULL) {
      node->prev->next = to_rm->next;
   }
   else {
      fprintf(stderr, "Erro: tentou-se remover o dummy head node da lista.\n");
      exit(EXIT_FAILURE);
   }

   free(to_rm);
   return payload;
}


//Remover um no especifico da lista
void *list_remove_pos(List *head, int pos)
{
   List *to_rm = list_follow(head, pos);
   void *payload = list_remove(to_rm);
   return payload;
}
```

```
#ifndef LIST_H
#define LIST_H

// Doubly-linked list with dummy head nodes
struct List {
   void *payload; // payload genérica
   struct List *next;
   struct List *prev;
};
```

```c
typedef struct List List;

List *list_follow(List *head, int pos);
void list_insert_pos(List *head, int pos, void *payload);
void list_append(List *head, void *payload);
void *list_remove(List *node);
void *list_remove_pos(List *head, int pos);

#endif
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <stdbool.h>
#include "logic.h"
#include "list.h"
#include "file.h"
#include "sdl.h"
#include "ea.h"
#include "error.h"

// Definição das operaçes válidas nas pilhas de cartas
void stack_push(Stack **sp, Card *card)
{
   Stack *old_sp = *sp;
   Stack *new = (Stack *) ecalloc((size_t) 1, sizeof(Stack));

   new->card = card;

   new->next = old_sp;
   new->prev = NULL;
   if (old_sp != NULL)
      old_sp->prev = new;
   else {
      // estamos a pushar o primeiro elemento, sp tava a NULL
   }

   *sp = new;
}

Card *stack_pop(Stack **sp)
{
   if (*sp == NULL) {
      fprintf(stderr, "Erro: tentou-se fazer pop numa stack vazia.\n");
      exit(EXIT_FAILURE);
   }

   Stack *pop = *sp;
   Card *card = pop->card;
   *sp = pop->next;
   if (pop->next != NULL)
      pop->next->prev = NULL;

   free(pop);

   return card;
}

int init_game(Config *config, List *players)
{
   Player *new_player = NULL;
   const int num_decks = config->num_decks;
```

```
   for (int i = 0; i < MAX_PLAYERS; i++) {
      new_player = (Player *) ecalloc((size_t) 1, sizeof(Player));
      if (i + 1 <= config->num_players) {
         // Jogadores efetivos
         new_player->type = config->player_type[i];
         new_player->ingame = true;
         strcpy(new_player->name, config->player_names[i]);
         new_player->money = config->money[i];
         new_player->orig_bet = config->bets[i];
         new_player->bet = new_player->orig_bet;
      }
      else {
         // Lugar não especificado na configuração.
         strcpy(new_player->name, "Empty");
         new_player->type = VA;
         new_player->ingame = false;
      }
      new_player->playing = false;
      list_append(players, new_player);
   }

   free(config);

   return num_decks;
}

void give_card(Player *player, Megadeck *megadeck)
{
   int random = 0;

   if (megadeck->cards_left == 0)
      megadeck->cards_left = create_megadeck(megadeck);

   // random: 1 - cards_left
   // random éo nmero de ns a seguir na lista, por isso,
   // tem de ser pelo menos 1 (dummy head node),
   // ou no máximo o nmero de ns (se seguirmos *cards_left
   //  ns a partir do dummy head node, chegamos  tail)
   random = rand() % megadeck->cards_left + 1;
   List *random_node = megadeck->deck;
   for (int i = 0; i < random; i++) {
      if (random_node->next != NULL)
         random_node = random_node->next;
      else {
         fprintf(stderr, "Erro: tentou-se dar uma carta não existente.\n");
         exit(EXIT_FAILURE);
      }
   }

   count_cards((Card *) random_node->payload, megadeck);

   stack_push(&(player->cards), random_node->payload);
   player->num_cards++;
   list_remove(random_node);
   megadeck->cards_left--;
}

int create_megadeck(Megadeck *megadeck)
{
   int total_cards = 0;
   Card *cur_card = NULL;

   for (int i = 0; i < megadeck->num_decks; i++)
      for (int j = 0; j < 4; j++)
         for (int k = 0; k < SUIT_SIZE; k++) {
```

```
            cur_card = (Card *) ecalloc(1, sizeof(Card));
            cur_card->suit = j;
            cur_card->id = k;
            list_append(megadeck->deck, cur_card);
          }

    total_cards = megadeck->num_decks * DECK_SIZE;
    return total_cards;
}

void new_game(List *players, Player *house, Megadeck *megadeck)
{
    //  s  fazer new_game quando já toda a gente jogou
     if (find_active_player(players) != NULL) {
       return;
    }

     //  s  fazer new_game quando houver jogadores para jogar
     if (find_ingame_player(players) == NULL) {
         return;
     }

    // atualizar as contagens das EAs com os valores da ronda anterior,
    // antes de qualquer carta ser distribuda na nova ronda.
    update_count(players, megadeck); // soma player->count com megadeck->count

    // Limpar cartas e retirar apostas
    clear_cards_take_bet(players, house, megadeck);
    // Dar cartas
    distribute_cards(players, house, megadeck);
    // Encontrar qual o jogador que começa a jogar
    find_playing(players, house);

    // Se a casa tiver blackjack...
    if (house->status == BJ) {
       // ...a ronda acaba logo.
       pay_bets(players, house);
       return;
    }
}

// limpar cartas e retirar apostas aos jogadores que possam jogar
void clear_cards_take_bet(List *players, Player *house, Megadeck *megadeck)
{
     List *aux = players->next;
     Player *cur_player = NULL;
     while (aux != NULL) {
       cur_player = (Player *) aux->payload;
       if (cur_player->money < cur_player->bet)
          cur_player->ingame = false;

       // Se o jogador jogou na ronda antes
       if (cur_player->ingame) {
          // Limpar cartas antigas
          destroy_stack(&cur_player->cards);
          cur_player->num_cards = 0;

          // Verificar se o jogador pode jogar outra vez e
          // retirar as apostas a todos os jogadores
          // (apenas fazemos o cálculo dos dinheiros no final da ronda!)
          if (cur_player->type == HU) {
             if (cur_player->money < cur_player->bet)
                cur_player->ingame = false;
             else
                cur_player->money -= cur_player->bet;
```

```
        }
        else if (cur_player->type == EA) {
           // O jogador éEA, chamar hi_lo para modificar
           // a sua aposta antes de esta ser retirada,
           // de acordo com a estratégia hi-lo.
           #ifdef HI_LO
           hi_lo(cur_player, megadeck);
           #endif
           cur_player->money -= cur_player->bet;
        }
      }
      aux = aux->next;
   }
   destroy_stack(&house->cards);
}

void distribute_cards(List *players, Player *house, Megadeck *megadeck)
{
   List *aux = players->next;
   Player *cur_player = NULL;
   int public_house_points = 0;

   // Distribuir cartas realisticamente
    for (int i = 0; i < 2; i++) {
      aux = players->next;
      cur_player = NULL;
      while (aux != NULL) {
         cur_player = (Player *) aux->payload;
         // se puder jogar...
         if (cur_player->ingame)
            // dar uma carta
            give_card(cur_player, megadeck);
         aux = aux->next;
      }
      if (i == 1) // segunda vez
         public_house_points = count_points(house);
      give_card(house, megadeck);
   }

   house->num_cards = 1; // desenhar s uma carta
   count_points(house);
    if (house->points == 21)
      house->status = BJ;
   else {
      house->points = public_house_points;
      house->status = WW;
   }
}

void find_playing(List *players, Player *house)
{
    bool found = false;
    List *aux = players->next;
    Player *cur_player = NULL;
   while (aux != NULL) {
      cur_player = (Player *) aux->payload;
      if (cur_player->ingame) {
         // Colocar status a Waiting
         cur_player->status = WW;

         // Contar pontos e verificar se tem blackjack
         count_points(cur_player);
         if (cur_player->points == 21)
            cur_player->status = BJ;
```

```
        // Dar a vez ao primeiro jogador sem blackjack
        // E não dar a vez a ninguém se a casa tiver blackjack
        if (!(house->status == BJ) && !(cur_player->status == BJ) && !found) {
           cur_player->playing = true;
           found = true;
        }
        else
           cur_player->playing = false;

     }
     aux = aux->next;
   }
}

List *find_ingame_player(List *players)
{
   List *aux = players->next; // dummy head
   Player *cur_player = NULL;
   while (aux != NULL) {
      // iterar até ao jogador que está a jogar
      cur_player = (Player *) aux->payload;
      if (cur_player->ingame)
         break;
      else
         aux = aux->next;
    }

    return aux;
}

List *find_active_player(List *players)
{
   List *aux = players->next; // dummy head
   Player *cur_player = NULL;
   while (aux != NULL) {
      // iterar até ao jogador que está a jogar
      cur_player = (Player *) aux->payload;
      if (cur_player->playing)
         break;
      else
         aux = aux->next;
    }

    return aux;
}

/*
 * Esta função échamada na main apenas para
 * registar as teclas de hit, stand, etc. se o jogador
 * for humano (ou seja para as teclas não afetarem as
 * jogadas dos jogadores EA mesmo se primidas)
 */
List *find_active_human_player(List *players)
{
   List *aux = players->next; // dummy head
   Player *cur_player = NULL;
   while (aux != NULL) {
      // iterar até ao jogador que está a jogar
      cur_player = (Player *) aux->payload;
      if (cur_player->playing && cur_player->type == HU)
         break;
      else
         aux = aux->next;
    }
```

```
    return aux;
}

void surrender(List *players, Player *house, Megadeck *megadeck)
{
    List *aux = find_active_player(players);
    if (aux == NULL) {
      return;
   }

    Player *cur_player = (Player *) aux->payload;

    cur_player->status = SU;
    stand(players, house, megadeck);
}

bool double_bet(List *players, Player *house, Megadeck *megadeck)
{
    List *aux = find_active_player(players);
    // não fazer nada se não for a vez dum jogador
    if (aux == NULL) {
        return false;
    }

    Player *cur_player = (Player *) aux->payload;

   // não fazer nada se o jogador não pode fazer double
    if (cur_player->money < cur_player->bet || cur_player->num_cards != 2)
        return false;

    cur_player->money -= cur_player->bet;
    cur_player->bet += cur_player->bet;

    player_hit(players, house, megadeck);
    if (!(cur_player->status == BU)) {
        stand(players, house, megadeck);
   }

   return true;
}

void bet(List *players)
{
    List *aux = find_active_player(players);
    if (aux != NULL) {
      return;
   }

   get_new_bet(players);
}

// TODO: Se o jogador adicionado for EA, subtrair megadeck->round_count
// a player->count, para quando for somado em new_game(),
// este dar zero (para esta EA começar a jogar efetivamente quando entrou,
// em vez de saber os dados de count da ronda anterior).
AddPlayerError add_player(List *players, List *old_players, SDL_Window *window)
{
   int pos = get_clicked_player();
   if (pos == 0) {
     return OUT;
   }

   List *aux = list_follow(players, pos);
   Player *old_player = (Player *) aux->payload;
   if (old_player->ingame) {
```

```
      return NOTEMPTY;
   }

   show_add_player_input_message(window);

   Player *new_player = get_new_player(pos);
   // subtrair a count da ronda para, quando somarmos em new_game,
   // ficar a zero.
   //if (new_player->type == EA)
   // new_player->count -= megadeck->count;

   old_player = (Player *) list_remove_pos(players, pos);
   list_append(old_players, old_player);
   list_insert_pos(players, pos, new_player);

   return OK;
}

void stand(List *players, Player *house, Megadeck *megadeck)
{
   List *aux = find_active_player(players);
   Player *cur_player = NULL;
   bool end_of_round = false;

   // Se não encontrarmos um jogador a jogar...
   if (aux == NULL) {
      // não fazer nada
        return;
    }

   // se encontrarmos, fazer-lhe stand
   cur_player = (Player *) aux->payload;
   if (cur_player->status == WW)
      cur_player->status = ST;
   cur_player->playing = false;

   // passar ao prximo jogador
   aux = aux->next;

   if (aux != NULL) {
      // se este prximo jogador existir,
      // procurar o prximo jogador válido a seguir
      while (aux != NULL) {
         cur_player = (Player *) aux->payload;
         if (cur_player->ingame && !(cur_player->status == BJ))
            break;
         else
            aux = aux->next;
      }
      // se ele existir, dar-lhe a vez
      if (aux != NULL) {
         cur_player->playing = true;
      }
      else {
         // não existe um prximo jogador válido para jogar
         end_of_round = true;
      }
   }
   else {
      // se não existir um prximo jogador, fizemos stand do ltimo jogador
      end_of_round = true;
   }

   if (end_of_round) {
      house_hit(house, megadeck);
```

```
      pay_bets(players, house);
   }
}

void player_hit(List *players, Player *house, Megadeck *megadeck)
{
   List *aux = find_active_player(players);
   Player *cur_player = NULL;

   if (aux != NULL) {
      cur_player = (Player *) aux->payload;
   }
   else {
      return;
   }

   give_card(cur_player, megadeck);
   count_points(cur_player);

   if (cur_player->points > 21) {
      cur_player->status = BU;
   }

   if (cur_player->points >= 21) {
      stand(players, house, megadeck);
   }
}

void house_hit(Player *house, Megadeck *megadeck)
{
   house->num_cards = 2;

   while (house->points <= 16) {
      give_card(house, megadeck);
      count_points(house);
   }

   if (house->points > 21) {
      house->status = BU;
   }
}


// Define o estado do jogador
// Atualiza a aposta no fim do jogo
void pay_bets(List *players, Player *house)
{
    List *aux = players->next;
   Player *cur_player = NULL;
   while (aux != NULL) {
        cur_player = ((Player *) aux->payload);

       // not playing
       if (!cur_player->ingame) {
           // skip this player
           aux = aux->next;
           continue;
       }

     // surrender
     if (cur_player->status == SU) {
        house->money -= cur_player->bet / 2;
        cur_player->money += cur_player->bet / 2;
         cur_player->losses++;
       }
```

```
      // blackjack casa e do jogador: tie
        else if (cur_player->status == BJ && house->status == BJ) {
            cur_player->money += cur_player->bet;
            cur_player->ties++;
        }
        // blackjack do jogador: win
        else if (cur_player->status == BJ && !(house->status == BJ)) {
         cur_player->money += 2*cur_player->bet + cur_player->bet/2;
            house->money -= cur_player->bet + cur_player->bet/2;
            cur_player->wins++;
        }
        // blackjack da casa: loss
        else if (!(cur_player->status == BJ) && house->status == BJ) {
         house->money += cur_player->bet;
         cur_player->losses++;
        }
        // bust da casa e do jogador: loss
        else if (cur_player->status == BU) {
            house->money += cur_player->bet;
            cur_player->losses++;
        }
        // bust da casa: win
        else if (!(cur_player->status == BU) && house->status == BU) {
            cur_player->money += 2*cur_player->bet;
         house->money -= cur_player->bet;
            cur_player->wins++;
        }
        // empate mesmos pontos: tie
        else if (cur_player->points == house->points) {
            cur_player->money += cur_player->bet;
            cur_player->ties++;
        }
        // jogador ganha com mais pontos: win
        else if (cur_player->points > house->points) {
            cur_player->money += 2*cur_player->bet;
            house->money -= cur_player->bet;
            cur_player->wins++;
        }
        // house ganha com mais pontos: loss
        else if (cur_player->points < house->points) {
         house->money += cur_player->bet;
         cur_player->losses++;
        }
        else {
         // isto nunca pode acontecer
         fprintf(stderr, "Erro: estado de jogador desconhecido.\n");
         exit(EXIT_FAILURE);
        }
        aux = aux->next;
    }
}


// Conta os pontos do jogador
// No primeiro loop atribui a todos os ases 11 pontos
// No segundo loop se pontos>21 remove 10 pontos ate não haverem ases
int count_points(Player *player)
{
    Stack *cards = player->cards;
    int num_ace = 0;

   player->points = 0;
    while (cards != NULL) {
        player->points += point_index(cards->card->id);
        if (cards->card->id == 12)
```

```
            num_ace++;
        cards = cards->next;
    }

    while (player->points > 21 && num_ace > 0) {
        player->points -= 10;
        --num_ace;
    }

    return player->points;
}


// Conversão do id da carta para pontos
int point_index(int id)
{
    int points = 0;
    id %= 13;
    if (id == 12)
        points = 11;
    else if (id <12 && id>8)
        points = 10;
    else
        points = id + 2;
    return points;
}

void destroy_list(List *head)
{
   List *aux = head->next; // dummy head
   List *tmp = NULL;
   while (aux != NULL) {
      tmp = aux;
      aux = tmp->next;
      free(tmp->payload);
      free(tmp);
   }
   free(head);
}

void destroy_stack(Stack **cards)
{
   while (*cards != NULL)
      free(stack_pop(cards));
}

void destroy_players_list(List *players)
{
    List *aux = players->next;
    Player *cur_player = NULL;
    while (aux != NULL) {
      cur_player = (Player *) aux->payload;
      destroy_stack(&cur_player->cards);
      aux = aux->next;
    }
   destroy_list(players);
}
```

```
#ifndef LOGIC_H
#define LOGIC_H

#include <stdbool.h>
#include <SDL2/SDL.h>
#include "types.h"
```

```c
#include "list.h"

#define MAX_LINE_LENGTH 64
#define DECK_SIZE 52
#define SUIT_SIZE 13

int init_game(Config *config, List *players);

void stack_push(Stack **sp, Card *card);
Card *stack_pop(Stack **sp);

void give_card(Player *player, Megadeck *megadeck);
int create_megadeck(Megadeck *megadeck);

void new_game(List *players, Player *house, Megadeck *megadeck);
void clear_cards_take_bet(List *players, Player *house, Megadeck *megadeck);
void distribute_cards(List *players, Player *house, Megadeck *megadeck);
void find_playing(List *players, Player *house);

List *find_ingame_player(List *players);
List *find_active_human_player(List *players);
List *find_active_player(List *players);

void surrender(List *players, Player *house, Megadeck *megadeck);
bool double_bet(List *players, Player *house, Megadeck *megadeck);
void bet(List *players);
AddPlayerError add_player(List *players, List *old_players, SDL_Window *window);
void stand(List *players, Player *house, Megadeck *megadeck);
void player_hit(List *players, Player *house, Megadeck *megadeck);
void house_hit(Player *house, Megadeck *megadeck);

void pay_bets(List *players, Player *house);
int count_points(Player *player);
int point_index(int id);

void destroy_list(List *head);
void destroy_stack(Stack **cards);
void destroy_players_list(List *players);

#endif
```

```c
/* Projecto de Programação
 * Implementação em C do jogo de casino Blackjack
 * Autores:
 * João Pinheiro:
 * João Freitas: joao.m.freitas@tecnico.ulisboa.pt
 */



#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include <SDL2/SDL_image.h>
#include <time.h>
#include "main.h"
#include "logic.h"
#include "file.h"
#include "sdl.h"
#include "ea.h"
#include "error.h"
```

```
// TODO: COMENTAR CDIGO COMO DEVE DE SER!
// TODO: Adicionar "autores" e introdução e etc.!
int main(int argc, char *argv[])
{
    SDL_Window *window = NULL;
    SDL_Renderer *renderer = NULL;
    TTF_Font *serif = NULL;
    SDL_Surface *cards[MAX_DECK_SIZE+1] = {0};
    SDL_Surface *imgs[2] = {0};
    SDL_Event event;
    int delay = 300;
   int ea_delay = 1500;
    bool quit = false;
    bool add_player_key = false;
    AddPlayerError add_player_error = OK;

   if (argc != 3) {
      fprintf(stderr, "Erro: nmero inválido de argumentos.\n");
      puts("Utilização:");
      printf("%s <ficheiro de config. do jogo> <ficheiro de config. das EAs>\n", argv[0]);
      exit(EXIT_FAILURE);
   }

   // Ler ficheiro de configuração dos jogadores
   Config *config = read_config(argv[1]);

   // Ler ficheiro de estrategia das EAs
   Strategy *strategy = read_strategy(argv[2]);

   // Declarar a lista de jogadores
   List *players = (List *) ecalloc((size_t) 1, sizeof(List));
   // enchê-la com dados do ficheiro de configuração
   const int num_decks = init_game(config, players);

   // Declarar a lista de jogadores velhos
   List *old_players = (List *) ecalloc((size_t) 1, sizeof(List));

   // Inicializar o megabaralho
   // é uma struct que contém a lista das cartas em si
    int cards_left = 0;
   List *deck = (List *) ecalloc((size_t) 1, sizeof(List));
    Megadeck megadeck_real = {cards_left, num_decks, deck, 0};
    Megadeck *megadeck = &megadeck_real;
    megadeck->cards_left = create_megadeck(megadeck);

   // Inicializar a casa
   Player *house = (Player *) ecalloc((size_t) 1, sizeof(Player));
   // Inicializar um novo jogo
    srand(time(NULL));
   new_game(players, house, megadeck);

    // loads the cards images
    LoadCards(cards);

   // initialize graphics
   InitEverything(WIDTH_WINDOW, HEIGHT_WINDOW, &serif, imgs, &window, &renderer);

   List *aux = find_active_player(players);
   bool ea = false;
   while (!quit) {
        // while there's events to handle
        while (SDL_PollEvent(&event)) {
         if (event.type == SDL_QUIT) {
               // user killed the window
```

```
            quit = true;
        }
        else if (event.type == SDL_KEYDOWN) {
            switch (event.key.keysym.sym) {
                case SDLK_q:
                    if (find_active_player(players) == NULL ||
                        find_ingame_player(players) == NULL)
                        quit = true;
                    break;

                case SDLK_n:
                    new_game(players, house, megadeck);
                    break;

                case SDLK_r:
                    if (find_active_human_player(players) != NULL)
                        surrender(players, house, megadeck);
                    break;

                case SDLK_d:
                    if (find_active_human_player(players) != NULL)
                        double_bet(players, house, megadeck);
                    break;

                case SDLK_b:
                    bet(players);
                    break;

                case SDLK_a:
                    if (find_active_player(players) == NULL)
                        add_player_key = true;
                    break;

                case SDLK_s:
                    if (find_active_human_player(players) != NULL)
                        stand(players, house, megadeck);
                    break;

                case SDLK_h:
                    if (find_active_human_player(players) != NULL)
                        player_hit(players, house, megadeck);
                    break;

                case SDLK_UP:
                    ea_delay+=100;
                    break;

                case SDLK_DOWN:
                    if (ea_delay > 100)
                        ea_delay-=100;
                    break;

                default:
                    break;
            }
        }
    }

    if (add_player_key)
        show_add_player_message(window);
      // render game table
      RenderTable(players, serif, imgs, renderer);
      // render house cards
      RenderHouseCards(house, cards, serif, renderer);
      // render player cards
```

```
      RenderPlayerCards(players, cards, renderer);
      // render colorful status rects above player
    render_status(players, serif, renderer);
      // render in the screen all changes above
      SDL_RenderPresent(renderer);
      // add a delay
    SDL_Delay(delay);

    aux = find_active_player(players);
    ea = false;
    if (aux != NULL)
       if (((Player * ) aux->payload)->type == EA)
          ea = true;

    if (ea) {
       ea_make_decision(players, house, megadeck, strategy);
       SDL_Delay(ea_delay);
    }

    else if (add_player_key) {
       add_player_error = add_player(players, old_players, window);
       if (add_player_error != OK)
          show_add_player_error_message(window, add_player_error);
       add_player_key = false;
    }
   }

  write_stats(players, house, old_players);

  destroy_players_list(players);
  destroy_players_list(old_players);
   destroy_stack(&house->cards);
  free(house);
  destroy_list(megadeck->deck);
  destroy_matrix(strategy->hard, HARD_LINES);
  destroy_matrix(strategy->soft, SOFT_LINES);
  free(strategy);

   UnLoadCards(cards);
   TTF_CloseFont(serif);
   SDL_FreeSurface(imgs[0]);
   SDL_FreeSurface(imgs[1]);
   SDL_DestroyRenderer(renderer);
  SDL_DestroyWindow(window);
  SDL_Quit();

  return EXIT_SUCCESS;
}
```

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include <SDL2/SDL_image.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include "main.h"
#include "sdl.h"

const char myName1[] = "João Pinheiro 84086";
const char myName2[] = "João Freitas 84093";

/**
 * RenderTable: Draws the table where the game will be played, namely:
```

```
 * - some texture for the background
 * - the right part with the IST logo and the student name and number
 * - squares to define the playing positions of each player
 * - names and the available money for each player
 * \param _money amount of money of each player
 * \param _img surfaces where the table background and IST logo were loaded
 * \param _renderer renderer to handle all rendering in a window
 */
void RenderTable(List *players, TTF_Font *_font, SDL_Surface *_img[], SDL_Renderer *_renderer)
{
    SDL_Color black = {0, 0, 0, 255}; // black
    SDL_Texture *table_texture;
    SDL_Rect tableSrc, tableDest;
    int height;
    char money_str[STRING_SIZE];

    // set color of renderer to white
    SDL_SetRenderDrawColor(_renderer, 255, 255, 255, 255);

    // clear the window
    SDL_RenderClear(_renderer);

    tableDest.x = tableSrc.x = 0;
    tableDest.y = tableSrc.y = 0;
    tableSrc.w = _img[0]->w;
    tableSrc.h = _img[0]->h;

    tableDest.w = SEP;
    tableDest.h = HEIGHT_WINDOW;

    table_texture = SDL_CreateTextureFromSurface(_renderer, _img[0]);
    SDL_RenderCopy(_renderer, table_texture, &tableSrc, &tableDest);

    // render the IST Logo
    height = RenderLogo(SEP, 0, _img[1], _renderer);

    // render the student name
    height += RenderText(SEP+3*MARGIN, height, myName1, _font, &black, _renderer);

    // this renders the student number
    height += RenderText(SEP+3*MARGIN, height, myName2, _font, &black, _renderer);

    // 2xnewline
    height += 2*RenderText(SEP+3*MARGIN, height, " ", _font, &black, _renderer);

   List *aux = players->next;
   Player *cur_player = NULL;
   while (aux) {
      cur_player = (Player *) aux->payload;
      if (cur_player->ingame) {
         sprintf(money_str, "%s (%s): %d euros",
               cur_player->name, cur_player->type == HU ? "HU" : "EA", cur_player->money);
         height += RenderText(SEP+3*MARGIN, height, money_str, _font, &black, _renderer);
      }
      aux = aux->next;
   }

   RenderPlayerArea(players, _renderer, _font);

    // destroy everything
    SDL_DestroyTexture(table_texture);
}


/* Desenhar a area do jogador
```

```
 * Nome, aposta, estado e pontos
 * Quadrado de cor diferente para o jogador que esta a jogar
 */
void RenderPlayerArea(List *players, SDL_Renderer* _renderer, TTF_Font *_font)
{
    SDL_Color white = {255, 255, 255, 255};
    SDL_Rect playerRect;
    char points_str[STRING_SIZE];
    char status_str[STRING_SIZE];
    List *aux = players->next;
    Player *cur_player = NULL;
    int num_player = 0;

   while (aux) {
      cur_player = (Player *) aux->payload;
      if (cur_player->ingame) {
         if (cur_player->playing)
            SDL_SetRenderDrawColor(_renderer, 255, 0, 0, 255);
         else
            SDL_SetRenderDrawColor(_renderer, 255, 255, 255, 255);

         playerRect.x = num_player*PLAYER_RECT_X;
         playerRect.y = PLAYER_RECT_Y;
         playerRect.w = PLAYER_RECT_W;
         playerRect.h = PLAYER_RECT_H;

         if (cur_player->status == WW || cur_player->status == ST)
            sprintf(points_str, "%d", cur_player->points);
         else if (cur_player->status == BJ)
            sprintf(points_str, "BJ");
         else if (cur_player->status == BU)
            sprintf(points_str, "BU");
         else if (cur_player->status == SU)
            sprintf(points_str, "SU");

         sprintf(status_str, "%s -- bet: %d, points: %s",
               cur_player->name, cur_player->bet, points_str);
         RenderText(playerRect.x, playerRect.y-30, status_str, _font, &white, _renderer);
         SDL_RenderDrawRect(_renderer, &playerRect);
      }
      aux = aux->next;
      num_player++;
    }
}


void show_add_player_message(SDL_Window *window)
{
   SDL_ShowSimpleMessageBox(SDL_MESSAGEBOX_INFORMATION,
                     "Adicionar Jogador",
                     "Clique num lugar vazio para inserir um novo jogador.",
                     window);
}

void show_add_player_error_message(SDL_Window *window, AddPlayerError error)
{
   char error_msg[MAX_STR_SIZE] = {0};

   switch(error) {
      case OUT:
         strcpy(error_msg, "Nao clicou dentro da area dos jogadores.\n"
               "Tente novamente primindo a tecla <a>.");
         break;

      case NOTEMPTY:
```

```
        strcpy(error_msg, "Nao selecionou um lugar vazio.\n"
             "Tente novamente primindo a tecla <a>.");
        break;

     default:
        break;
   }
   SDL_ShowSimpleMessageBox(SDL_MESSAGEBOX_INFORMATION,
                    "Adicionar Jogador",
                    error_msg,
                    window);
}

void show_add_player_input_message(SDL_Window *window)
{
   SDL_ShowSimpleMessageBox(SDL_MESSAGEBOX_INFORMATION,
                    "Adicionar Jogador",
                    "Insira os dados do jogador no terminal.",
                    window);
}


//Obter posição para inserir o novo jogador
int get_clicked_player()
{
   SDL_Event event;
   int i = 0;

   while (1) {
      SDL_PollEvent(&event);
      if (event.type == SDL_MOUSEBUTTONDOWN)
         break;
   }

   int mouse_x = event.button.x;
   int mouse_y = event.button.y;

   if (mouse_y >= PLAYER_RECT_Y && mouse_y <= PLAYER_RECT_Y + PLAYER_RECT_H)
      while (mouse_x >= 0*PLAYER_RECT_X) {
         mouse_x -= PLAYER_RECT_W;
         i++;
      }
   else
      i = 0;

   return i;
}

/**
 * RenderHouseCards: Renders cards of the house
 * \param _house vector with the house cards
 * \param _pos_house_hand position of the vector _house with valid card IDs
 * \param _cards vector with all loaded card images
 * \param _renderer renderer to handle all rendering in a window
 */
void RenderHouseCards(Player *house, SDL_Surface **_cards, TTF_Font *_font, SDL_Renderer* _renderer)
{
    int x = 0, y = 0;
    int div = WIDTH_WINDOW/CARD_WIDTH;
   Card *cur_card = NULL;
   int card_id = 0;
   int num_cards = 0;
    SDL_Color white = { 255, 255, 255, 255};
   char status_str[STRING_SIZE] = {0};
   char points_str[STRING_SIZE] = {0};
```

```
   if (house->status == WW || house->status == ST)
      sprintf(points_str, "%d", house->points);
   else if (house->status == BJ)
      sprintf(points_str, "BJ");
   else if (house->status == BU)
      sprintf(points_str, "BU");

   sprintf(status_str, "dealer: %s points", points_str);
   RenderText(20, 130, status_str, _font, &white, _renderer);

    Stack *aux = house->cards;
    Stack *tmp = NULL;
    // drawing all house cards
   while (tmp != house->cards) {
      aux = house->cards;
        while (aux->next != tmp)
            aux = aux->next;

      cur_card = aux->card;
      card_id = cur_card->id + cur_card->suit * SUIT_SIZE;

        // calculate its position
        x = (div/2 - house->num_cards/2 + num_cards)*CARD_WIDTH + 15;
        y = (int) (0.26f*HEIGHT_WINDOW);
        RenderCard(x, y, card_id, _cards, _renderer);

      num_cards++;
        tmp = aux;
    }

    // If the dealer has only 2 cards and no blackjack, draw the second card face down
    if (house->num_cards == 1 && house->status != BJ) {
        x = (div/2-house->num_cards/2+1)*CARD_WIDTH + 15;
        y = (int) (0.26f*HEIGHT_WINDOW);
        RenderCard(x, y, MAX_DECK_SIZE, _cards, _renderer);
    }
}

/**
 * RenderPlayerCards: Renders the hand, i.e. the cards, for each player
 * \param _player_cards 2D array with the player cards, 1st dimension is the player ID
 * \param _pos_player_hand array with the positions of the valid card IDs for each player
 * \param _cards vector with all loaded card images
 * \param _renderer renderer to handle all rendering in a window
 */
void RenderPlayerCards(List *players, SDL_Surface **_cards, SDL_Renderer* _renderer)
{
    int pos = 0, x = 0, y = 0;
    int num_player = 0;
    int num_cards = 0;
    int card_id = 0;

    List *aux = players->next; // dummy head
    Player *cur_player = NULL;
    Card *cur_card = 0;
   Stack *aux_cards = NULL;
    // Iterate over all players
    while (aux) {
        cur_player = (Player *) aux->payload;
      if (cur_player->ingame) {
         // Iterate over the stack backwards
         aux_cards = cur_player->cards;
         if (aux_cards)
            while (aux_cards->next)
```

```
            aux_cards = aux_cards->next;

        // agora aux_cards aponta para o ltimo elemento da stack
        while (aux_cards) {
           // get the card
           cur_card = aux_cards->card;
           card_id = cur_card->id + cur_card->suit * SUIT_SIZE;

           // draw the card
           pos = num_cards % 4;
           x = (int) num_player * (SEP/4-5)+(num_cards/4)*12+15;
           y = (int) PLAYER_RECT_Y+10;
           if ( pos == 1 || pos == 3) x += CARD_WIDTH + 30;
           if ( pos == 2 || pos == 3) y += CARD_HEIGHT+ 10;
           RenderCard(x, y, card_id, _cards, _renderer);

           num_cards++;
           aux_cards = aux_cards->prev;
        }
        num_cards = 0;
     }
     aux = aux->next;
     num_player++;
    }
}

/**
 * RenderCard: Draws one card at a certain position of the window, based on the card code
 * \param _x X coordinate of the card position in the window
 * \param _y Y coordinate of the card position in the window
 * \param _num_card card code that identifies each card
 * \param _cards vector with all loaded card images
 * \param _renderer renderer to handle all rendering in a window
 */
void RenderCard(int _x, int _y, int _num_card, SDL_Surface **_cards, SDL_Renderer* _renderer)
{
    SDL_Texture *card_text;
    SDL_Rect boardPos;

    // area that will be occupied by each card
    boardPos.x = _x;
    boardPos.y = _y;
    boardPos.w = CARD_WIDTH;
    boardPos.h = CARD_HEIGHT;

    // render it !
    card_text = SDL_CreateTextureFromSurface(_renderer, _cards[_num_card]);
    SDL_RenderCopy(_renderer, card_text, NULL, &boardPos);

   // destroy everything
    SDL_DestroyTexture(card_text);
}

/**
 * LoadCards: Loads all images of the cards
 * \param _cards vector with all loaded card images
 */
void LoadCards(SDL_Surface **_cards)
{
   int i = 0;
    char filename[STRING_SIZE] = {0};

     // loads all cards to an array
    for (i = 0; i < MAX_DECK_SIZE; i++) {
        // create the filename !
```

```
        sprintf(filename, ".//assets//cartas//carta_%02d.png", i+1);
        // loads the image !
        _cards[i] = IMG_Load(filename);
        // check for errors: deleted files ?
        if (_cards[i] == NULL) {
            fprintf(stderr, "Unable to load image: %s\n", SDL_GetError());
            exit(EXIT_FAILURE);
        }
    }
    // loads the card back
    _cards[i] = IMG_Load(".//assets//cartas//carta_back.jpg");
    if (_cards[i] == NULL) {
        fprintf(stderr, "Unable to load image: %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
    }
}


/**
 * UnLoadCards: unloads all card images of the memory
 * \param _cards vector with all loaded card images
 */
void UnLoadCards(SDL_Surface **_array_of_cards)
{
    // unload all cards of the memory: +1 for the card back
    for (int i = 0 ; i < MAX_DECK_SIZE + 1; i++ )
    {
        SDL_FreeSurface(_array_of_cards[i]);
    }
}
// Desenhar o estado do jogador
// Blackjack, Bust e Surrender
void render_status(List *players, TTF_Font *_font, SDL_Renderer *renderer)
{
    SDL_Rect rect;

    char bust[] = "BUST";
    char blackjack[] = "BLACKJACK";
   char surrender[] = "SURRENDER";

    List *aux = players->next;
    Player *cur_player = NULL;
    SDL_Color white = { 255, 255, 255, 255};
    for (int i=0; aux; i++) {
        cur_player = (Player *) aux->payload;
        rect.y = 380;
        rect.h = 30;
        if (cur_player->ingame) {
         if (cur_player->status == BJ) {
            rect.x = 55 + 208*i;
            rect.w = 115;
            SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255 );
            SDL_RenderFillRect(renderer, &rect);
            SDL_SetRenderDrawColor(renderer, 0, 0, 0, 255 );
            SDL_RenderDrawRect(renderer, &rect);
            RenderText(64+208*i, 382, blackjack, _font, &white, renderer);
         }
         else if (cur_player->status == BU) {
            rect.x = 80 + 208*i;
            rect.w = 70;
            SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255 );
            SDL_RenderFillRect(renderer, &rect);
            SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255 );
            SDL_RenderDrawRect(renderer, &rect);
            RenderText(94+(208*i), 382, bust, _font, &white, renderer);
```

```
        }
        else if (cur_player->status == SU) {
           rect.x = 55 + 208*i;
           rect.w = 115;
           SDL_SetRenderDrawColor(renderer, 255, 200, 0, 255 );
           SDL_RenderFillRect(renderer, &rect);
           SDL_SetRenderDrawColor(renderer, 255, 200, 0, 255 );
           SDL_RenderDrawRect(renderer, &rect);
           RenderText(64+208*i, 382, surrender, _font, &white, renderer);
        }

     }
       aux = aux->next;
    }
}

/**
 * RenderLogo function: Renders the IST Logo on the window screen
 * \param x X coordinate of the Logo
 * \param y Y coordinate of the Logo
 * \param _logoIST surface with the IST logo image to render
 * \param _renderer renderer to handle all rendering in a window
 */
int RenderLogo(int x, int y, SDL_Surface *_logoIST, SDL_Renderer* _renderer)
{
   SDL_Texture *text_IST;
   SDL_Rect boardPos;

    // space occupied by the logo
   boardPos.x = x;
   boardPos.y = y;
   boardPos.w = _logoIST->w;
   boardPos.h = _logoIST->h;

    // render it
   text_IST = SDL_CreateTextureFromSurface(_renderer, _logoIST);
   SDL_RenderCopy(_renderer, text_IST, NULL, &boardPos);

    // destroy associated texture !
   SDL_DestroyTexture(text_IST);
   return _logoIST->h;
}

/**
 * RenderText function: Renders the IST Logo on the window screen
 * \param x X coordinate of the text
 * \param y Y coordinate of the text
 * \param text string where the text is written
 * \param font TTF font used to render the text
 * \param _renderer renderer to handle all rendering in a window
 */
int RenderText(int x, int y, const char *text, TTF_Font *_font, SDL_Color *_color, SDL_Renderer* _renderer)
{
   SDL_Surface *text_surface;
   SDL_Texture *text_texture;
   SDL_Rect solidRect;

   solidRect.x = x;
   solidRect.y = y;
    // create a surface from the string text with a predefined font
   text_surface = TTF_RenderUTF8_Blended(_font,text,*_color);
   if (text_surface == NULL)
   {
       fprintf(stderr, "TTF_RenderText_Blended: %s\n", TTF_GetError());
       exit(EXIT_FAILURE);
```

```
   }
    // create texture
   text_texture = SDL_CreateTextureFromSurface(_renderer, text_surface);
    // obtain size
   SDL_QueryTexture( text_texture, NULL, NULL, &solidRect.w, &solidRect.h );
    // render it !
   SDL_RenderCopy(_renderer, text_texture, NULL, &solidRect);

   SDL_DestroyTexture(text_texture);
   SDL_FreeSurface(text_surface);
   return solidRect.h;
}

/**
 * InitEverything: Initializes the SDL2 library and all graphical components: font, window, renderer
 * \param width width in px of the window
 * \param height height in px of the window
 * \param _img surface to be created with the table background and IST logo
 * \param _window represents the window of the application
 * \param _renderer renderer to handle all rendering in a window
 */
void InitEverything(int width, int height, TTF_Font **_font, SDL_Surface *_img[], SDL_Window** _window,
     SDL_Renderer** _renderer)
{
    InitSDL();
    InitFont();
    *_window = CreateWindow(width, height);
    *_renderer = CreateRenderer(width, height, *_window);

    // load the table texture
    _img[0] = IMG_Load("assets//table_texture.png");
    if (_img[0] == NULL) {
        fprintf(stderr, "Unable to load image: %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
    }

    // load IST logo
    _img[1] = SDL_LoadBMP("assets//ist_logo.bmp");
    if (_img[1] == NULL) {
        fprintf(stderr, "Unable to load bitmap: %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
    }

    // this opens (loads) a font file and sets a size
    *_font = TTF_OpenFont("assets//FreeSerif.ttf", 16);
    if(*_font == NULL) {
        fprintf(stderr, "TTF_OpenFont: %s\n", TTF_GetError());
        exit(EXIT_FAILURE);
    }
}

/**
 * InitSDL: Initializes the SDL2 graphic library
 */
void InitSDL()
{
    // init SDL library
   if (SDL_Init(SDL_INIT_EVERYTHING) != 0) {
      fprintf(stderr, "Failed to initialize SDL: %s\n", SDL_GetError());
        exit(EXIT_FAILURE);
   }
}

/**
 * InitFont: Initializes the SDL2_ttf font library
```

```
 */
void InitFont()
{
   // Init font library
   if (TTF_Init() == -1) {
       fprintf(stderr, "TTF_Init: %s\n", TTF_GetError());
        exit(EXIT_FAILURE);
   }
}

/**
 * CreateWindow: Creates a window for the application
 * \param width width in px of the window
 * \param height height in px of the window
 * \return pointer to the window created
 */
SDL_Window *CreateWindow(int width, int height)
{
    SDL_Window *window;
    // init window
   window = SDL_CreateWindow("Blackjack", WINDOW_POSX, WINDOW_POSY, width+EXTRASPACE, height, 0);
    // check for error !
   if (window == NULL) {
      fprintf(stderr, "Failed to create window : %s\n", SDL_GetError());
      exit(EXIT_FAILURE);
   }

   return window;
}

/**
 * CreateRenderer: Creates a renderer for the application
 * \param width width in px of the window
 * \param height height in px of the window
 * \param _window represents the window for which the renderer is associated
 * \return pointer to the renderer created
 */
SDL_Renderer *CreateRenderer(int width, int height, SDL_Window *_window)
{
    SDL_Renderer *renderer;
    // init renderer
   renderer = SDL_CreateRenderer(_window, -1, 0);

   if (renderer == NULL) {
      fprintf(stderr, "Failed to create renderer : %s", SDL_GetError());
        exit(EXIT_FAILURE);
   }

   // set size of renderer to the same as window
   SDL_RenderSetLogicalSize(renderer, width+EXTRASPACE, height);

   return renderer;
}
```

```
#ifndef SDL_H
#define SDL_H

#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include <SDL2/SDL_image.h>
#include "main.h"
#include "logic.h"
#include "list.h"
```

```
#define STRING_SIZE 100    // max size for some strings
#define CARD_WIDTH 67      // card width
#define CARD_HEIGHT 97     // card height
#define WINDOW_POSX 100    // initial position of the window: x
#define WINDOW_POSY 100    // initial position of the window: y
#define WIDTH_WINDOW 900   // window width
#define HEIGHT_WINDOW 525  // window height
#define EXTRASPACE 150
#define MARGIN 5
#define SEP ((int) (0.95f*WIDTH_WINDOW))
#define PLAYER_RECT_X (SEP/4 - 5) + 10
#define PLAYER_RECT_Y ((int) (0.55f*HEIGHT_WINDOW))
#define PLAYER_RECT_W SEP/4 - 5
#define PLAYER_RECT_H ((int) (0.42f*HEIGHT_WINDOW))

void InitEverything(int width, int height, TTF_Font **_font, SDL_Surface *_img[], SDL_Window** _window,
     SDL_Renderer** _renderer);
void InitSDL();
void InitFont();

SDL_Window* CreateWindow(int width, int height);
SDL_Renderer* CreateRenderer(int width, int height, SDL_Window *_window);

int RenderText(int x, int y, const char *text, TTF_Font *_font, SDL_Color *_color, SDL_Renderer* _renderer);
void RenderPlayerArea(List *players, SDL_Renderer* _renderer, TTF_Font *_font);

void show_add_player_message(SDL_Window *window);
void show_add_player_error_message(SDL_Window *window, AddPlayerError error);
void show_add_player_input_message(SDL_Window *window);

int get_clicked_player();

int RenderLogo(int, int, SDL_Surface *, SDL_Renderer *);

void RenderTable(List *players, TTF_Font *_font, SDL_Surface *_img[], SDL_Renderer *_renderer);

void RenderHouseCards(Player *house, SDL_Surface **_cards, TTF_Font *_font, SDL_Renderer* _renderer);
void RenderPlayerCards(List *players, SDL_Surface **_cards, SDL_Renderer* _renderer);
void RenderCard(int _x, int _y, int _num_card, SDL_Surface **_cards, SDL_Renderer* _renderer);

void LoadCards(SDL_Surface **_cards);
void UnLoadCards(SDL_Surface **_array_of_cards);

void render_status(List *players, TTF_Font *_font, SDL_Renderer *renderer);

// definition of some strings: they cannot be changed when the program is executed !
extern const char myName[];
extern const char myNumber[];

#endif
```

```
#define MAX_PLAYER_NAME 8
#define MAX_PLAYERS 4
#include "list.h"

// Entidade Artificial, Humano, Vazio
typedef enum {EA, HU, VA} Type;

// waiting, blackjack, busted, stand'ed', surrender
typedef enum {WW, BJ, BU, ST, SU} Status;

// Okay, user clicked outside playerarea, user clicked on valid player
typedef enum {OK, OUT, NOTEMPTY} AddPlayerError;
```

```
// hit, stand, surrender, double->hit, double->stand
typedef enum {H='H', S='S', R='R', D='D', E='E'} Move;

typedef struct Card {
   int suit;
   int id;
} Card;

typedef struct Stack {
   Card *card;
   struct Stack *next;
   struct Stack *prev;
} Stack;

typedef struct Player {
   Type type;
   char name[MAX_PLAYER_NAME+1];
   bool ingame;
   bool playing;
    Status status;
   int money;
   int bet;
   int orig_bet;
   Stack *cards;
   int num_cards;
   int points;
   int wins;
   int losses;
   int ties;
   // guarda a contagem das cartas para o hi-lo no caso do jogador ser EA
   int count;
} Player;

typedef struct Config {
   int num_decks;
   int num_players;
   int player_type[MAX_PLAYERS];
   char player_names[MAX_PLAYERS][MAX_PLAYER_NAME+1];
   int money[MAX_PLAYERS];
   int bets[MAX_PLAYERS];
} Config;

typedef struct Megadeck {
    int cards_left;
    const int num_decks;
    List *deck;
   // guarda a contagem das cartas duma ronda para o hi-lo das EAs
   int round_count;
} Megadeck;

typedef struct Strategy {
    Move **hard;
    Move **soft;
} Strategy;
```