DCC/ICEx/UFMG — Gleison Mendonça e Luigi Soares
DCC 203 - Programação e Desenv. de Software I — **Exame Especial - Turmas TA1, TA2, TM3**
2º semestre de 2022 — 20/12/2022

Estudante:	**Matrícula:**

Instruções:

i. Este exame possui 5 página(s). Verifique se sua cópia do exame está completa.

ii. **Esta prova é sem consulta**: você não tem permissão para consultar o livro texto, suas notas de aula, a Internet, seus colegas ou quaisquer outras pessoas ou fontes para concluir o exame.

iii. Se você acredita que alguma pergunta esteja subespecificada, anote as suposições que você teve que fazer para chegar a sua resposta e justifique-as como parte de sua resposta à pergunta.

iv. **Lembre-se de indentar o código** de maneira apropriada e atente-se a organização, clareza e legibilidade do código!

Nesta prova, vamos construir um software para gerenciar um banco. Os dados de cada cliente estarão armazenados em um arquivo CSV que obedece o seguinte formato (cada linha corresponde a um cliente):

```
1 Nome;Saldo;Limite;Ativo 1;Ativo 2;Ativo 3
2 Shakira;400.10;4000.00;A440.00;F7000.12;T300.00
3 David Guetta;1000.10;900.00;A00.44;T120.00;F10540.00
4 Jose Rico;100.10;900.00;F0.00;A4493240.04;T500.00
```

1. (14 pontos) É necessário processar um arquivo contendo os clientes, suas respectivas contas e os ativos financeiros em seu nome. Os ativos financeiros podem ser ações, títulos públicos ou fundos imobiliários. Os ativos financeiros adotam os seguintes códigos de identificação: **A** — Ações; **T** — Títulos Públicos; **F** — Fundos Imobiliários.

   (a) (4 pontos) Defina a estrutura para criar um objeto do tipo **Conta**. Sua estrutura deve conter os campos **saldo** (double) e **limite** (double).

   (b) (4 pontos) Defina a estrutura para armazenar um **Ativo**. Sua estrutura deve conter os campos **tipo** (char) e **valor** (double).

   (c) (6 pontos) Defina a estrutura para armazenar um **Cliente**. Sua estrutura deve conter os campos **nome** (string), **ativos** (vetor do tipo Ativo com 3 posições), e **Conta** (ponteiro para valor do tipo Conta).

banco.h

```
1  #ifndef BANCO_H_
2  #define BANCO_H_
3
4  typedef struct Conta {
5      double saldo;
6      double limite;
7  } Conta;
8
9  typedef struct Ativo {
10     char tipo; // 'A', 'T', 'F'
11     double valor;
12 } Ativo;
```

banco.h

```
11 typedef struct Cliente {
12     char *nome;
13     Ativo ativos[3];
14     Conta *conta;
15 } Cliente;
16
17 Cliente processa_cliente(char *linha);
18 int processa_arquivo(char *nome_arquivo,
19                       Cliente **clientes);
20 int f(Cliente *clientes, int n, int k,
21       int i, Cliente *r);
22 #endif
```

2. (30 pontos) Considere a seguinte estrutura **ResultadoProcValor**, que contém dois campos: **valor** (o valor processado) e **tamanho** (a quantidade de caracteres lidas).

banco.c

```
1 typedef struct ResultadoProcValor {
2     double valor; unsigned int tamanho;
3 } ResultadoProcValor;
```

Atenção: Nas questões a seguir, as única funções pré-definidas que você pode utilizar são **strlen**, **malloc** e **realloc**.

(a) (15 pontos) Implemente uma função **processa_valor**, que recebe como parâmetro uma string contendo um número real, *potencialmente* seguido de um ponto e vírgula (delimitador utilizado no arquivo CSV). Esta função deve processar o número real presente no início da string e retornar um valor do tipo **ResultadoProcValor**, onde o campo **valor** é o número que foi processado e o campo **tamanho** é a quantidade de caracteres lidos. Por exemplo, se a entrada for "23.15;10.23": apenas o primeiro número será processado, então o valor será 23.15 e a quantidade de caracteres consumidos foi 5. Sua função deve ter a seguinte assinatura:

```
ResultadoProcValor processa_valor(char *linha);
```

Resposta:

banco.c

```
#define DELIMITADOR ';'
ResultadoProcValor processa_valor(char *linha) {
    double inteiro = 0.0;
    // Assume que o separador de decimal eh um "ponto".
    // 400.15:
    // inteiro = 0 * 10 + 4 = 4
    // inteiro = 4 * 10 + 0 = 40
    // inteiro = 40 * 10 + 0 = 400
    int i;
    for (i = 0; linha[i] != '.'; i++) {
        inteiro = inteiro * 10 + (linha[i] - '0');
    }
    double decimal = 0.0;
    double div = 10.0;
    // decimal = 0 + 1 / 10 = 0.1
    // decimal = 0.1 + 5/100 = 0.1 + 0.05 = 0.15
    for (i++; linha[i] != DELIMITADOR && linha[i] != '\0'; i++) {
        decimal += (linha[i] - '0') / div;
        div *= 10;
    }

    ResultadoProcValor r;
    r.valor = inteiro + decimal;
    r.tamanho = i;

    return r;
}
```

(b) (15 pontos) Implemente a função **processa_cliente** declarada no cabeçalho `banco.h`. Esta função recebe como parâmetro uma string que corresponde a uma linha do arquivo CSV, e retorna um valor do tipo **Cliente**. Assuma que a string não contém `\n` no final. Se atente aos campos que devem ser alocados dinamicamente.

Resposta:

banco.c

```
Cliente processa_cliente(char *linha) {
    Cliente c;

    // Nome do cliente:
    int i;
    c.nome = (char *)malloc(150 * sizeof(char));
    for (i = 0; linha[i] != DELIMITADOR; i++) {
        c.nome[i] = linha[i];
    }
    c.nome = (char *) realloc(c.nome, (strlen(c.nome) + 1) * sizeof(char));

    i++; // Pula delimitador

```

```
14      // Conta do cliente:
15      c.conta = (Conta *)malloc(sizeof(Conta));
16
17      ResultadoProcValor r = processa_valor(&linha[i]);
18      c.conta->saldo = r.valor;
19
20      i += r.tamanho; // Move para o delimitador
21      i++; // Pula o delimitador
22
23      r = processa_valor(&linha[i]);
24      c.conta->limite = r.valor;
25
26      i += r.tamanho; // Move para o delimitador
27
28      // Ativos do cliente:
29      for (int j = 0; linha[i] != '\0'; j++) {
30          i++; // Pula o delimitador
31          c.ativos[j].tipo = linha[i];
32
33          i++;
34          r = processa_valor(&linha[i]);
35          c.ativos[j].valor = r.valor;
36
37          i += r.tamanho; // Move para o delimitador ou final da linha
38      }
39
40      return c;
41  }
```

3. (26 pontos) A função **processa_arquivo** percorre o arquivo CSV, lendo linha por linha e armazenando os clientes lidos em um vetor. Ela recebe como parâmetro o nome do arquivo (uma string), que deve ser aberto em modo *leitura*. O segundo parâmetro é um ponteiro para o vetor de clientes que será alocado e preenchido (note que o vetor já é um ponteiro, configurando assim um ponteiro para outro ponteiro; então, para acessar o vetor precisamos passar pelo primeiro ponteiro: `*clientes`). A função retorna a quantidade de clientes processados. Você deve completá-la nos locais indicados. Vamos assumir que cada linha do arquivo tem, no máximo, 150 caracteres.

banco.c

```
1   #define TAMANHO_MAX_LINHA 150
2   int processa_arquivo(char *nome_arquivo, Cliente **clientes) {
3       // Abra o arquivo em modo leitura:
4       FILE *fp = fopen(nome_arquivo, "r");
5       if (fp == NULL) return -1;
6
7       int n = 0; int capacidade = 10;
8
9       // Aloque espaco para 10 clientes (utilize a variavel capacidade):
10      *clientes = (Cliente *)malloc(capacidade * sizeof(Cliente));
11      char linha[TAMANHO_MAX_LINHA + 1];
12
13      // A primeira linha do arquivo eh o cabecalho com o nome de cada coluna. Ela
14      // nao sera utilizada, mas precisamos passar por ela. Pule a primeira linha:
15      fgets(linha, TAMANHO_MAX_LINHA + 1, fp);
16
17      // Agora leia a segunda linha, que potencialmente eh um cliente:
18      fgets(linha, TAMANHO_MAX_LINHA + 1, fp);
19
20      // Como saber se chegamos ao final do arquivo? Indique a condicao:
21      while (!feof(fp)) {
22          if (linha[strlen(linha) - 1] == '\n') linha[strlen(linha) - 1] = '\0';
23
```

```
24          Cliente c = processa_cliente(linha);
25          if (n >= capacidade) {
26              capacidade *= 2;
27              // Realoque o vetor de clientes de acordo com a nova capacidade:
28              *clientes = (Cliente *)realloc(*clientes, capacidade * sizeof(Cliente));
29          }
30
31          (*clientes)[n] = c; n++;
32
33          // Leia a proxima linha:
34          fgets(linha, TAMANHO_MAX_LINHA + 1, fp);
35      }
36      // Feche o arquivo:
37      fclose(fp);
38      return n;
39  }
```

4. (30 pontos) A função **f** a seguir foi implementada de forma recursiva. Ela assume que o vetor **r** passado por parâmetro já foi alocado e tem tamanho **n**, o mesmo tamanho que o vetor **clientes**.

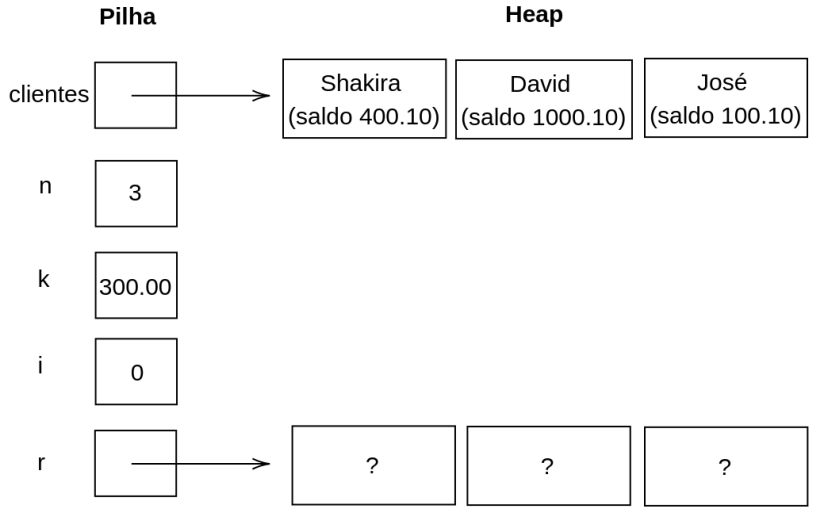
banco.c

```
1  int f(Cliente *clientes, int n, int k, int i, Cliente *r) {
2      if (n == 0) {
3          return 0;
4      }
5
6      if (clientes[n - 1].conta->saldo >= k) {
7        r[i] = clientes[n - 1];
8        return 1 + f(clientes, n - 1, k, i + 1, r);
9      }
10
11     return f(clientes, n - 1, k, i, r);
12 }
```

Uma possível chamada inicial a esta função seria:

```
int m = f(clientes, n, 300.00, 0, r);
```

Descreva sucintamente o que a função faz. Indique qual o caso base e qual o caso recursivo. Dê exemplos da execução da função, mostrando o estado da pilha a cada chamada. Você não precisa mostrar todos os parâmetros e campos de estruturas detalhadamente, apenas os relevantes. Segue um exemplo do estado inicial da pilha, após a primeira chamada da função (assuma que a pilha cresce de cima para baixo):



Resposta: A função **f** percorre todos os clientes, selecionando e armazenando no vetor **r** apenas aqueles clientes com saldo $\geq$ **k**. A função, além de preencher o vetor **r** com os clientes filtrados, retorna a quantidade de clientes que satisfazem a condição saldo $\geq$ **k**. O caso base ocorre quando **n** == 0, isto é, a quantidade de clientes restantes a serem analisados é zero; nesta situação, o valor retornado é zero. No caso recursivo, se o n-ésimo cliente (i.e. índice $n-1$) possui saldo $\geq$ **k**, ele é adicionado ao vetor e é somado 1 ao resultado da chamada recursiva, para indicar que este cliente foi selecionado; caso contrário, o resultado é o retorno da própria chamada recursiva.

Considerando a pilha inicial acima, na 1ª chamada o n-ésimo cliente é o José Rico, cujo saldo não é maior que **k** = 300.00. Logo, ele não será adicionado ao vetor e o retorno da função será o resultado da chamada recursiva. Na segunda chamada (**n** = 2), o cliente David será adicionado ao vetor e o resultado será 1 somado ao resultado da próxima chamada recursiva. Na 3ª chamada (**n** = 1), a cliente Shakira também será adicionada ao vetor e o resultado será 1 somado ao resultado da chamada recursiva. Neste ponto, o vetor **r** contém os clientes David e Shakira. Na 4ª e última chamada, **n** == 0 e o resultado é zero. Quando as chamadas retornam para o ponto inicial, o resultado é $0+1+1+0=2$ clientes selecionados (esperava-se que o aluno mostrasse a pilha a cada chamada).