Prova 3 - 2021/01

Ao concordar em fazer esta prova, eu juro que seguirei o **código de honra**:

1. Não ajudarei e nem pedirei ajuda a ninguém durante o período de submissão do exame;
2. Não divulgarei qualquer informação sobre as minhas soluções durante o período de submissão do exame.

É normal não entender completamente uma questão. Se você tiver dúvidas sobre qualquer questão, envie uma mensagem ***PRIVADA*** no Teams ou um e-mail para olmo@dcc.ufmg.br com o assunto **[PROVA 3]** e explique a sua dúvida para mim. Tentarei responder o mais rápido possível, mas a resposta pode demorar se a mensagem for enviada fora do horário comercial. **Não responderei mensagens enviadas depois das 23:59 do dia 26/08. Não responderei mensagens enviadas através do sistema do Moodle!**

Importante #1: assim que iniciar a sua solução no VPL, um arquivo com `prova3.c` será criado para você. O arquivo já vem com *includes*, *defines*, *tipos de dados* e *funções* que serão usadas ao longo da prova. Não modifique os **nomes** desses recursos.

Importante #2: você pode usar qualquer função de bibliotecas padrões da linguagem C (o que inclui a math.h) e criar funções e recursos adicionais sempre que precisar.

Importante #3: Além dos testes automáticos, também farei avaliações manuais rápidas. Por exemplo, vou zerar todas as questões recursivas que usarem loops ou variáveis globais. Questões de alocações dinâmicas que fizerem uso de alocação estática também serão zeradas.

Questões

1) Considere um arquivo que registra o saldo final diário de cinco filiais do mesmo restaurante. Cada linha representa um dia e cada coluna uma das filiais. Implemente uma função **para somar todos os saldos diários de uma dada filial entre um dia inicial até um dia final (inclusive para ambos).** A sua função recebe quatro argumentos, na seguinte ordem: (1) uma ***string*** como parâmetro contendo o nome do arquivo que deve ser lido (ex: "`soma.txt`"); (2) o índice da filial (ex: 5), o dia inicial (ex: 7) e o dia final (ex: 10). As filiais são numeradas de 1 (saldos registrados na primeira coluna) a 5 (saldos registrados na última coluna). Os dias são contados a partir da primeira linha do arquivo, que representa o dia 1, e vão até a última linha do arquivo, **que pode ter um número de linhas arbitrariamente grande**. Considere que todas as filiais terão sempre os mesmos dias registrados no arquivo, ou seja, todas as linhas terão sempre cinco colunas, um saldo para cada filial. Cada coluna é separada da outra por um caracter de espaço, conforme o exemplo a seguir:

```
53.55 140.43 -48.86 -39.00 47.42
```

Se qualquer parâmetro for inválido, a sua função deve retornar 0.

Casos em que os parâmetros são inválidos: (1) o arquivo não existe; (2) o índice da filial não está entre 1 e 5; (3) o dia final é menor que o dia inicial. Caso o dia inicial seja menor que 1, faça a soma começar da primeira linha. Caso o dia final seja maior que o último dia registrado no arquivo, faça a soma até a última linha do arquivo. Para esta prova, o arquivo `saldo_franquias.txt` está disponível para você, contendo 10 linhas. Exemplo: para esse arquivo, se `filial` for **5**, `ini` for **7** e `fim` for **10**, a sua função deve retornar **327.31**. Lembre que outros arquivos podem ser usados, com qualquer número de linhas. Protótipo:

```
float somaSaldo(char nome_arquivo[], int filial, int ini, int fim);
```

Valor da questão: **7** pontos

2) Seu avô está precisando de ajuda para organizar as finanças de casa e mantê-las arquivadas para controlar seus gastos mensais. Para tal, você criou uma lista de gastos e a armazenou em um arquivo. Neste exercício você vai criar uma função para gerenciar estes gastos.

Implemente uma função **para carregar uma lista de gastos para a memória.** A sua função recebe uma ***string*** como parâmetro, contendo o nome do arquivo que deve ser lido, e retorna o ponteiro para a lista na memória. A **primeira linha do arquivo** contém o número de meses da lista e **cada linha subsequente** representa um mês, com o mês de referência, o ano, a quantidade de gastos, e os gastos separados por espaços. A lista de gastos de cada mês deve ser armazenada como um vetor de pontos flutuantes em uma matriz "dinâmica", em que as linhas devem ter a quantidade **exata** de colunas para armazenar a

lista e um sinalizador final, que deve ser o número `0.00`. Por exemplo, se há **5** colunas no arquivo, a linha da matriz correspondente deve ter **6**, uma para cada coluna e uma final para o `0.00`. Além disso, sua matriz deve ter um sinalizador final para as linhas: se há **10** linhas de dados no arquivo, sua matriz deve ter **11**, uma para cada linha do arquivo e uma final para o sinalizador, que deve ser um ponteiro para o endereço nulo (**`NULL`**), indicando o fim da lista de gastos. Para esta prova, o arquivo lista.txt está disponível para você. A sua representação na memória deve ser feita como nesta tabela. Note que as linhas têm dimensões variáveis e que o ponto flutuante `0.00` delimita o fim de cada linha. Se o arquivo não existir, retorne o ponteiro nulo (**`NULL`**).

Dica: para ler o primeiro inteiro do arquivo e já saltar para a próxima linha usando **`fscanf`** faça `fscanf(arq, "%d\n", &num_linhas).`

Protótipo:

```
float** carregaLista(char nome_arquivo[]);
```

Valor da questão: **8** pontos

3) O seu avô quer agora saber o máximo que gastou em um mês. Para isso, você deve escrever uma função **RECURSIVA** para processar a matriz de gastos e retornar o valor total de gastos do mês que o seu avô gastou mais. A sua função recebe um ponteiro para a lista, que é representada exatamente como descrito no **exercício 2** desta prova. Para a lista armazenada no arquivo lista.txt, que foi carregado para a memória nesta forma, a sua função deve retornar **`836.44`**, que é o total de gastos em fevereiro de 2021, o mês em que seu avô gastou mais. Assuma que todos os gastos são maiores que zero. A sua função **não pode** usar *loops* (*for*, *while*, *goto*, etc) e nem variáveis globais. **Super dica que se você não a considerar com carinho você pode complicar demais a sua vida ao resolver esta questão:** você pode usar funções adicionais, desde que essas também não usem *loops* e variáveis globais.

Protótipo:

```
float maiorGasto(float **gastos);
```

Valor da questão: **5** pontos

4) Ao estudar os gastos armazenados, você e seu avô perceberam que ele estava precisando de um empréstimo. Ao contratar um empréstimo, as empresas cobram uma **taxa inicial**, que é somada ao valor inicial do empréstimo, e uma taxa de **juros** aplicada todo mês. Para ajudar seu avô a decidir em qual empresa ele deve pegar o empréstimo, escreva uma função **RECURSIVA** chamada **`valorTotalEmprestimo`**, que recebe um **valor** ponto flutuante que representa o valor do empréstimo, um inteiro **meses** que representa o número de meses até o pagamento do empréstimo, e uma instância de estrutura do tipo **Empresa** representando uma opção de empréstimo. **Empresas** são representadas pelo tipo de dados **`Empresa`**, descrito abaixo, que possui dois campos: um do tipo `float`,

denominado **juros**, que representa o valor de juros a ser aplicado mensalmente, e um do tipo `float` chamado **taxa**, que representa a taxa inicial do empréstimo.

```
typedef struct Empresa {
  float juros; //1% == 0.01
  float taxa;
}Empresa;
```

A sua função deve retornar o valor total cobrado pela empresa ao final do empréstimo. O valor da taxa inicial deverá ser somada antes do cálculo do juros e **o valor será pago integralmente após o último mês, ou seja, não deverão ser deduzidas parcelas mensais do valor**. Juros são compostos, ou seja, são aplicados ao final de cada mês sobre o valor acumulado devido.

Exemplo: se o valor do empréstimo for de **R$90,00**, o número de meses para pagar o empréstimo for de **3 meses**, a taxa inicial for de **R$10,00**, e a taxa de juros for de **1% (juros == 0.01)**, a sua função deve retornar **100.00** (valor do empréstimo + taxa inicial) **+ 1.00** (juros do primeiro para o segundo mês) **+ 1.01** (juros do segundo para o terceiro mês, do valor acumulado, que será 101.00) **+ 1.0201** (juros do terceiro para o quarto mês, do valor acumulado, que será 102.01) **= 103.0301**.

Para este exercício (e para facilitar), considere que os valores de todos os parâmetros (incluindo os valores da Empresa) são estritamente maiores que zero.

Sua função **não pode** usar *loops* (*for*, *while*, *goto*, etc) e nem variáveis globais. Note que, caso o seu avô se preocupasse mais com as finanças e menos com recursividade e limitações de uso de loops, ele provavelmente não estaria nessa situação difícil. **Super dica que se você não a considerar com carinho você pode complicar demais a sua vida ao resolver esta questão:** você pode criar e usar funções adicionais, desde que essas também não usem *loops* e variáveis globais.

Protótipo:

```
float valorTotalEmprestimo(float valor, int m, Empresa empresa);
```

Valor da questão: **5** pontos

5) O seu avô adora programar, mas se complica demais com as chaves (`{` e `}`). Às vezes esquece de abrir uma chave, às vezes esquece de fechar. Para ajudá-lo na escrita de programas, você deve escrever uma função **RECURSIVA** para processar o programa que seu avô escreveu e verificar se ele esqueceu de abrir ou fechar alguma chave, e quantas chaves ele esqueceu. Se para cada chave abrindo há uma fechando, a sua função deve retornar 0. Caso o programa tenha **n** chaves abrindo que não foram fechadas, o seu programa deve retornar **n**. Por fim, caso o programa tenha **n** chaves fechando que não foram abertas, o seu programa deve retornar **-n**. Exemplo: para o programa abaixo, a sua função deve retornar **2**.

```
int main() {
      int i = 0;
      scanf(“%d”, &i);
      if(i == 0) {
            printf(“\nZero!”);
      else  {
            printf(“\nDiferente de zero!”);
      }
      printf(“\nFim!”);
```

Importante: se o seu avô escreveu um programa em que esqueceu de fechar uma chave e esqueceu de abrir uma outra, os dois erros se anulam e a sua função deve retornar **0**. Mas não se preocupe, você vai aprender a fazer isso corretamente quando fizer a disciplina de compiladores. :)

A sua função recebe uma string que armazena o programa como parâmetro e retorna essa diferença de chaves abrindo e fechando. Como de praxe, sua função **não pode** usar *loops* (*for*, *while*, *goto*, etc.) e nem variáveis globais.

Protótipo:

int diferencaChaves(char *programa);

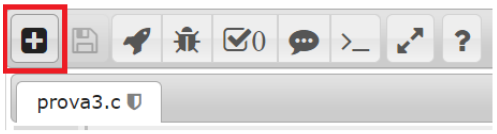
Valor da questão: **5** pontos

Execução no VPL

Para testar um exercício, inicie a execução no VPL e digite o número da função que gostaria de testar (**1**, **2**, **3, 4** ou **5**) e siga as instruções da tela.

Uma forma de testar as suas soluções no VPL é através do botão “Executar”. Para testar um exercício, inicie a execução no VPL e digite o número da função que gostaria de testar (**1**, **2**, **3**, **4** ou **5**). Depois, digite os argumentos da função.

Nas questões de arquivos, caso você queira utilizar um arquivo customizado para testes, insira ele na VPL, clicando no botão + e, em seguida, no ícone de papel, conforme a figura abaixo. Atente-se que só é possível inserir um arquivo adicional.



Para o **exercício 1**, digite também o nome do arquivo, a filial, a linha inicial e a linha final. Enquanto faz a prova você terá o arquivo `saldo_franquias.txt` disponível. Então, se quiser testar o exemplo do enunciado, digite:

```
1 saldo_franquias.txt 5 7 10
```

Para o **exercício 2**, digite também o **nome do arquivo** que você vai carregar na função. Enquanto faz a prova você terá o arquivo lista.txt disponível. Então, se quiser testar o exemplo do enunciado, digite:

```
2 lista.txt
```

Para o **exercício 3**, digite também o **nome do arquivo** que você vai carregar na função. Enquanto faz a prova você terá o arquivo lista.txt disponível. Então, se quiser testar o exemplo do enunciado, digite:

```
3 lista.txt
```

Para o **exercício 4**, digite também o **valor do empréstimo**, o **número de meses**, os **juros** e a **taxa inicial**. Então, se quiser testar o exemplo do enunciado, digite:

```
4 90 3 0.01 10
```

Para o **exercício 5**, digite também o **nome do arquivo** que você vai carregar na função. Enquanto faz a prova você terá o arquivo programa_avo.txt disponível. Então, se quiser testar o exemplo do enunciado, digite:

```
3 programa_avo.txt
```

Para essa prova, você pode copiar e colar códigos no VPL. Assim, se preferir, você pode implementar e testar as funções localmente (**com outros arquivos, que você mesmo pode criar**) e depois copiar e colar para o VPL. Não esqueça, no entanto, de compilar e testar todas as funções no VPL. Códigos que não compilam obtém nota **0**.

Outra forma de testar o seu programa é implementar a função `minha_main()`, que é chamada sempre que você digitar um número de exercício fora do intervalo **[1, 5]** (ex: **0**). A ideia é que esta função simule uma função **main** tradicional, e nela você pode fazer chamadas às funções que você implementou da maneira que desejar.

Os testes automáticos disponibilizados para esta prova não avaliam todos os aspectos das funções. Então, uma boa nota nos testes automáticos não garante uma boa nota final para a prova.