

UNIVERSIDADE VEIGA DE ALMEIDA
BACHAREL EM CIÊNCIA DA COMPUTAÇÃO

LUCAS RODRIGUES CORREIA - 1240105219

LUCAS FERRAZ DE PAULA SILVA - 1240114448

VINÍCIUS TAVARES – 1240114051

VITOR HUGO FRIAS SILVA DOS SANTOS - 1240207416

SISTEMA DE CONTROLE DE CONTA

RIO DE JANEIRO

2025

LUCAS RODRIGUES CORREIA

LUCAS FERRAZ DE PAULA SILVA

VINÍCIUS TAVARES

VITOR HUGO FRIAS SILVA DOS SANTOS

SISTEMA DE CONTROLE DE CONTA

Trabalho apresentado à disciplina
Algoritmo e laboratório de Programação.

Orientador: Prof. Denis Gonçalves Colpe

RIO DE JANEIRO

2025

SUMÁRIO

1 - Introdução	4
2 - Estrutura do Projeto	5
2.1 - Diretórios Principais.....	5
2.2 - Exemplo de implementação	5
3 - Funcionalidade do Sistema	6
3.1 Tela de menu	6
3.2 Tela de Cadastro de Conta.....	8
3.3 Tela de Lançamento de Movimentação	11
3.4 Tela de Remoção de movimentação	14
3.5 Tela de Extrato e Saldo de Conta	17
3.6 Tela de todas as movimentações por conta	20
3.7 Tela de Geração de Arquivo CSV	22
4 - Funções auxiliares e algoritmos utilizados	25
4.1 limpar_tela()	25
4.2 limpar_buffer()	25
4.3 CompararDatas(const char *data1, const char *data2)	26
4.4 removerItem(Conta *conta, int indice)	27
4.5 atualizarSaldoConta(Conta *conta, Movimentacao *mov)	27
4.6 reverterSaldoConta(Conta *conta, Movimentacao *mov).....	28
4.7 ordenarMovimentacoesPorData(Conta* conta).....	29
5 - Conclusão	30

1 - Introdução

No dia a dia, o uso de cartões de débito e crédito tornou-se uma prática comum nas transações financeiras. No entanto, essa convivência pode levar à perda de controle sobre os gastos realizados, dificultando a visualização precisa do consumo mensal. Visando resolver esse problema, foi desenvolvido um **sistema de controle de contas**, que permite ao usuário monitorar seus gastos de forma clara, prática e organizada.

O sistema oferece funcionalidades básicas e intuitivas. Inicialmente, é possível realizar o **cadastro de contas**, que podem ser do tipo débito ou crédito. Para contas de débito, o usuário pode ou não definir um saldo inicial, considerando a possibilidade de ser uma conta recém-criada. Já para contas de crédito, a definição de um valor inicial é obrigatória, representando o limite disponível do cartão.

Após a criação das contas, o sistema permite o **lançamento de movimentações**, que podem ser entradas ou saídas, sempre acompanhadas da **data da operação**. Além disso, o usuário pode especificar se a transação já foi efetivada ou se está programada para o futuro, como ocorre em compras parceladas ou agendadas.

Entre as funcionalidades disponíveis, destacam-se: a **consulta de saldo e extrato por período**, que facilita o acompanhamento dos gastos em intervalos específicos; a **visualização completa das movimentações por conta**, com o cálculo do saldo total; e a **persistência dos dados**, garantindo que as informações sejam mantidas entre diferentes sessões de uso.

Por fim, o sistema possibilita a **exportação dos dados em formato CSV**, permitindo que os registros sejam abertos em planilhas, como o Microsoft Excel, para uma análise mais detalhada e compartilhamento das informações.

2 - Estrutura do Projeto

A organização do projeto segue um padrão consolidado na engenharia de software, visando modularidade, manutenção e aderência a boas práticas. A estrutura de diretórios adotada é detalhada a seguir:

2.1 - Diretórios Principais

Diretório	Finalidade	Exemplo de conteúdo
bin/	Armazena executáveis e artefatos finais do sistema.	programa.exe, dados.csv
build/	Contém arquivos temporários gerados durante a compilação (objetos, logs).	main.obj , Makefile
include/	Centraliza cabeçalhos (.h, .hpp) para compartilhamento entre módulos.	config.h, biblioteca.hpp
src/	Agrupar o código-fonte principal, organizado por funcionalidades.	main.c , modulo/utilitarios

2.2 - Exemplo de implementação

```
.
└─ Sistema de conta/
    │
    └─ bin/
        │
        ├── Sistema_de_contas.exe
        ├── Sistema_de_Contas.csv
        └─ dados.bin
    │
    └─ build/
        │
        ├── faux.o
        ├── gerenciadorConta.o
        ├── main.o
        ├── menu.o
        └─ persistencia.o
    │
    └─ include/
        │
        ├── faux.h
        ├── gerenciadorConta.h
        ├── menu.h
        └─ persistencia.h
    │
    └─ src/
        │
        ├── faux.c
        ├── gerenciadorConta.c
        ├── main.c
        ├── menu.c
        └─ persistencia.c
```

Figura 1. árvore do projeto

3 - Funcionalidade do Sistema

Agora que conhecemos uma breve introdução sobre o funcionamento do sistema, vamos nos aprofundar em cada tela e seus respectivos códigos. O sistema foi desenvolvido utilizando o prompt de comando como interface gráfica e é composto por sete telas principais, cada uma responsável por uma funcionalidade específica. A primeira delas é a tela de menu, que serve como ponto de partida para as demais.

3.1 Tela de menu

A tela de menu funciona como a interface principal de navegação do sistema. Por meio dela, o usuário pode acessar todas as outras funcionalidades, como cadastro de contas, lançamento de movimentações, visualização de extratos, entre outras. O layout foi projetado para ser simples e intuitivo, facilitando o uso mesmo por pessoas sem familiaridade com sistemas computacionais.

```
=====
| Bem vindo ao SCC |
=====

1 - Cadastrar conta
2 - Lança movimentação
3 - Remover movimentação
4 - Exibir saldo e extrato
5 - Totalização de movimentações
6 - Gerar arquivo
0 - sair

Escolha uma das opções: _
```

Figura 2. Tela de menu

Para a criação do menu interativo, adotamos a estrutura de controle switch-case, que não apenas permite a navegação organizada e eficiente entre as diferentes funcionalidades do sistema, mas também, em conjunto com a estrutura de repetição do-while, garante que após a conclusão de qualquer operação, o sistema retorne ao menu principal, possibilitando a seleção de uma nova opção. A entrada de dados é realizada com a função scanf(), e a exibição das informações com printf(). As funcionalidades de limpar_tela, limpar_buffer e salvar_dados serão detalhadas em seguida. A implementação segue o seguinte fluxo::

```
// Abre um menu para utilizar as outras funções
void menu(){
    do {
        printf("=====\n");
        printf("| Bem vindo ao SCC | \n");
        printf("=====\n\n");

        printf("1 - Cadastrar conta\n");
        printf("2 - Lança movimentação\n");
        printf("3 - Remover movimentação\n");
        printf("4 - Exibir saldo e extrato\n");
        printf("5 - Totalização de movimentações\n");
        printf("6 - Gerar arquivo\n");
        printf("0 - sair\n");

        printf("Escolha uma das opções: ");
        scanf("%d", &opcao);
        limpar_buffer();

        switch (opcao){
            case 1:
                limpar_tela();
                cadastrarConta();
                salvar_dados("dados.bin"); // Salvando depois de cadastrar
                break;
            case 2:
                limpar_tela();
                lancaMovimentacao();
                salvar_dados("dados.bin"); // Salvando depois do lançamento
                break;
            case 3:
                limpar_tela();
                removerMovimentacao();
                salvar_dados("dados.bin"); // Salvando a remoção
                break;
            case 4:
                limpar_tela();
                exibirSaldoExtrato(); // Exibi o saldo e o Extato de uma Conta
                break;
            case 5:
                limpar_tela();
                totalMovimentacao(); // Exibi todas as movimentações de todas as Contas
                break;
            case 6:
                limpar_tela();
                gerarArquivo(); // Gerando o arquivo CSV
                limpar_tela();
                break;
            default :
                printf("\nOpção inválida!");
                limpar_buffer();
                limpar_tela();
        }
    } while(opcao != 0);
}
```

Figura 3. Código da tela de menu

3.2 Tela de Cadastro de Conta

Nesta tela, o usuário pode cadastrar novas contas no sistema, informando o tipo (débito ou crédito), nome da conta e valor inicial (opcional no caso de débito e obrigatório no caso de crédito). Essa etapa é fundamental para iniciar o uso do sistema e organizar os lançamentos conforme a origem dos recursos.

```
=====
| Cadastro de Conta |
=====

Nome da conta: Bradesco

Tipo da conta (D - débito | C - crédito): D

Deseja por um saldo inicial? (S - sim | N - não): S

Valor de entrada: 1000

Conta registrada com sucesso
```

Figura 4. Tela de cadastro

Para a criação desta tela, empregamos uma estrutura de dados (*struct*) em conjunto com *arrays* para o armazenamento eficiente dos dados inseridos. Adicionalmente, cada conta possui um identificador único (*id*), o que otimiza o gerenciamento nas etapas subsequentes do sistema. Complementarmente, utilizamos a estrutura de decisão *if-else* para direcionar o fluxo de acordo com o tipo de conta: para contas de débito, apresentamos uma opção para iniciar com saldo; para contas de crédito, o sistema solicita diretamente a definição do limite do cartão.


```

1 // Cadastra a conta e Adiona um Id a cada conta
2 void cadastrarConta(){
3     printf("=====\n");
4     printf(" | Cadastro de Conta | \n");
5     printf("=====\n\n");
6
7     // Id da conta
8     conta.id = id + 1;
9     id++;
10
11     printf("Nome da conta: ");
12     fgets(conta.nome, sizeof(conta.nome), stdin);
13     conta.nome[strcspn(conta.nome, "\n")] = '\0'; // Para remover o \n
14
15     do{
16         printf("\nTipo da conta (D - débito | C - crédito): ");
17         scanf(" %c", &conta.tipo);
18         limpar_buffer();
19     } while(conta.tipo != 'D' && conta.tipo != 'C');
20
21
22     if(conta.tipo == 'D'){
23         char resposta;
24         do{
25             printf("\nDeseja por um saldo inicial? (S - sim | N - não): ");
26             scanf("%c", &resposta);
27             limpar_buffer();
28         } while(resposta != 'S' && resposta != 'N');
29         if(resposta == 'S'){
30             printf("\nValor de entrada: ");
31             scanf("%f", &conta.valor);
32             limpar_buffer();
33         }
34     } else{
35         // Pois todo cartão de crédito possui um limite, então tem que incial com um
36         printf("\n Qual o limite do cartão? ");
37         scanf("%f", &conta.valor);
38         limpar_buffer();
39     }
40
41     contas[totalContas++] = conta;
42     printf("\nConta registrada com sucesso");
43     getchar();
44     limpar_tela();
45 }

```

Figura 5. Código da tela de cadastro

3.3 Tela de Lançamento de Movimentação

Através desta interface, o usuário pode adicionar novas movimentações financeiras à cota selecionada. Cada movimentação pode ser do tipo entrada (receita) ou saída (despesa), sendo possível informar também a data, o valor, uma descrição e se o lançamento é efetivo ou futuro. Essa funcionalidade permite o controle detalhado das operações realizadas.

```
=====
| Lançamento de movimentação |
=====

Contas encontradas:

ID - 1 | Nome - teste
ID - 2 | Nome - Bradesco

Digite o número do ID da conta: 2

Digite o nome da movimentação: Caderno

Digite o valor da movimentação: 10

Digite a data da movimentação: 23/01/2001

Digite o tipo da movimentação (E - efetivo | F - futuro): E

Foi entrada ou saída ? (E - entrada | S - saída): S

Movimentação realizada com sucesso_
```

Figura 6. Tela de movimentação

Para o registro de novas transações financeiras, esta tela emprega a função `printf()` para exibir um cabeçalho informativo. Inicialmente, verifica-se a existência de contas cadastradas (`totalContas`). Em caso negativo, informa-se ao usuário a necessidade de registrar uma conta previamente, interrompendo o fluxo com `getchar()`, `limpar_tela()` e `return`.

Havendo contas, estas são listadas, exibindo seu ID e Nome através de um loop `for`. Solicita-se então ao usuário o ID da conta para o lançamento, utilizando `scanf()` para leitura e `limpar_buffer()` para segurança.

Um novo loop for busca a conta correspondente ao ID inserido. Ao encontrar a conta:

1. Um espaço para a nova movimentação é alocado no *array* *movimentacoes* da conta.
2. Os dados da movimentação (nome, valor, data) são solicitados e armazenados com *scanf()* e *limpar_buffer()*.
3. A entrada para o tipo (E - efetivo /F - futuro) e o pagamento (E - entrada /S - saída) da movimentação são validados por loops do-while com *scanf()* e *limpar_buffer()*.
4. A função *atualizarSaldoConta()* é chamada para refletir a movimentação no saldo da conta.
5. Uma mensagem de sucesso é exibida, seguida por uma pausa com *getchar()*, limpeza da tela com *limpar_tela()* e retorno ao menu.

Caso o ID fornecido não seja encontrado após a busca, uma mensagem de erro é exibida, seguida por *getchar()* e *limpar_tela()* antes de retornar ao menu.

```

1  printf("=====\n");
2  printf("| Lançamento de movimentação |\n");
3  printf("=====\n\n");
4
5  if(totalContas == 0){
6      printf("Nenhuma conta encontrada. Registre uma conta primeiro!!!");
7      getchar();
8      limpar_tela();
9      return;
10 }
11
12 printf("Contas encontradas:\n");
13 for (int i = 0; i < totalContas; i++){
14     printf("\nID - %d | Nome - %s",contas[i].id, contas[i].nome);
15 }
16
17 int buscaId;
18 printf("\n\nDigite o número do ID da conta: ");
19 scanf("%d", &buscaId);
20 limpar_buffer();
21
22 for (int i = 0; i < totalContas; i++){
23     if(buscaId == contas[i].id){
24         Conta* conta = &contas[i];
25         Movimentacao* mov = &conta->movimentacoes[conta->totalMovimentacoes++];
26
27         printf("\nDigite o nome da movimentação: ");
28         scanf("%49s", mov->nome);
29         limpar_buffer();
30         printf("\nDigite o valor da movimentação: ");
31         scanf("%f", &mov->valor);
32         limpar_buffer();
33         printf("\nDigite a data da movimentação: ");
34         scanf(" %10s", mov->data);
35         limpar_buffer();
36
37         do{
38             printf("\nDigite o tipo da movimentação (E - efetivo | F - futuro): ");
39             scanf(" %c", &mov->tipo);
40             limpar_buffer();
41         } while(mov->tipo != 'E' && mov->tipo != 'F');
42
43         do{
44             printf("\nFoi entrada ou saída ? (E - entrada | S - saída): ");
45             scanf(" %c", &mov->pagamento);
46             limpar_buffer();
47         }while(mov->pagamento != 'E' && mov->pagamento != 'S');
48
49         // Logo após registra, ele atualia o saldo da conta
50         atualizarSaldoConta(conta, mov);
51
52         printf("\nMovimentação realizada com sucesso");
53         getchar();
54         limpar_tela();
55         return;
56     }
57 }
58
59 printf("\n\nID não encontrado.");
60 getchar();
61 limpar_tela();
62 }

```

Figura 7. Código da Tela de movimentação

3.4 Tela de Remoção de movimentação

Essa funcionalidade permite ao usuário excluir movimentações específicas que foram cadastradas incorretamente ou que não são mais relevantes. A exclusão é feita com base em filtros de conta, data e descrição, garantindo precisão e segurança na manipulação dos dados.

```
=====
| Remover Movimentação |
=====

Contas encontradas:

ID - 1 | Nome - teste
ID - 2 | Nome - Bradesco

Digite o número do ID da conta: 1

Id da movimentação: 1
Nome da movimentação: testando
Valor da movimentação: 100,00

Qual deseja remover ?
Informe o Id: 1

Removido com sucesso!!!_
```

Figura 8. Tela de Remover Movimentação

Para a exclusão de registros financeiros, esta tela inicia com a exibição de um cabeçalho formatado via `printf()`. Similar à tela de lançamento, verifica-se inicialmente a existência de contas (`totalContas`). Caso não haja contas, uma mensagem informativa é exibida, seguida por `getchar()`, `limpar_tela()` e `return` para retornar ao menu.

Existindo contas, estas são listadas com seus ID e Nome através de um loop for. O usuário é então solicitado a inserir o ID da conta da qual deseja remover uma movimentação, utilizando `scanf()` para leitura e `limpar_buffer()` para limpar o buffer de entrada.

Um loop for busca a conta correspondente ao ID fornecido. Ao encontrar a conta, o sistema verifica se existem movimentações registradas para essa conta (`contas[i].totalMovimentacoes`). Se não houver movimentações, uma mensagem informa o usuário, seguida por `getchar()`, `limpar_tela()` e `return`.

Caso existam movimentações, estas são listadas para o usuário, exibindo um "Id da movimentação" sequencial, o Nome e o Valor de cada uma através de outro loop for.

O usuário é então questionado sobre qual movimentação deseja remover, informando o seu "Id da movimentação". A entrada é lida com `scanf()` e o buffer é limpo com `limpar_buffer()`.

Antes da remoção, a função `reverterSaldoConta()` é chamada, passando a conta e a movimentação a ser removida (`contas[i].movimentacoes[idMov - 1]`). Esta função provavelmente ajusta o saldo da conta, desfazendo o efeito da movimentação que será excluída.

Em seguida, a função `removeItem()` é invocada, passando a conta e o índice da movimentação a ser removida (`idMov - 1`). Esta função deve implementar a lógica para remover o item do *array* de movimentações.

Após a remoção, uma mensagem de sucesso ("Removido com sucesso!!!") é exibida, seguida por `getchar()`, `limpar_tela()` e `return`.

Se o ID da conta fornecido não for encontrado após a busca, uma mensagem de erro ("ID não encontrado.") é exibida, seguida por `getchar()` e `limpar_tela()` antes de retornar ao menu.

```

1 void removerMovimentacao(){
2     printf("=====\n");
3     printf("| Remover Movimentação |\n");
4     printf("=====\n\n");
5
6     if(totalContas == 0){
7         printf("Nenhuma conta encontrada. Registre uma conta primeiro!!!");
8         getchar();
9         limpar_tela();
10        return;
11    }
12
13    printf("Contas encontradas:\n");
14    for (int i = 0; i < totalContas; i++){
15        printf("\nID - %d | Nome - %s",contas[i].id, contas[i].nome);
16    }
17
18    int buscaId;
19    printf("\n\nDigite o número do ID da conta: ");
20    scanf("%d", &buscaId);
21    limpar_buffer();
22
23    for (int i = 0; i < totalContas; i++){
24        if(buscaId == contas[i].id){
25            if (contas[i].totalMovimentacoes == 0) {
26                printf("Nenhuma movimentação registrada. Registre primeiro!!!");
27                getchar();
28                limpar_tela();
29                return;
30            }
31
32            for (int j = 0; j < contas[i].totalMovimentacoes; j++){
33                Movimentacao* mov = &contas[i].movimentacoes[j];
34                printf("\nId da movimentação: %d", j + 1);
35                printf("\nNome da movimentação: %s", mov->nome);
36                printf("\nValor da movimentação: %.2f", mov->valor);
37            }
38
39            int idMov;
40            printf("\n\nQual deseja remover ?");
41            printf("\nInforme o Id: ");
42            scanf("%d", &idMov);
43            limpar_buffer();
44
45            // Retorna o saldo antes da movimentação feita
46            Movimentacao* movRemover = &contas[i].movimentacoes[idMov - 1];
47            reverterSaldoConta(&contas[i], movRemover);
48
49            // Remove a movimentação do Array
50            removerItem(&contas[i], idMov - 1);
51
52            printf("\nRemovido com sucesso!!!");
53            getchar();
54            limpar_tela();
55            return;
56        }
57    }
58    printf("\n\nID não encontrado.");
59    getchar();
60    limpar_tela();
61 }

```

Figura 9. Código da tela de remoção de movimentação

3.5 Tela de Extrato e Saldo de Conta

Nesta tela, o usuário pode visualizar o saldo atual de uma conta específica e gerar o extrato com base em um intervalo de datas definido. Essa função é útil para verificar o comportamento financeiro ao longo de um período e identificar padrões de consumo.

```
=====
| Extrato e Saldo da conta |
=====

Contas encontradas:

ID - 1 | Nome - teste
ID - 2 | Nome - Bradesco

Digite o número do ID da conta: 2

Digite a data início da procura: 01/01/200
Digite a data final da procura: 31/12/2027

Movimentações no período 01/01/200 a 31/12/2027:

-----
Nome da movimentação: caderno
Valor: 10,00
Data: 01/02/2024
Entrada/Saída: S
Tipo da movimentação: E
=====
Saldo no período: -10,00
Saldo atual da conta é 990,00_
```

Figura 10. Tela de exibir extrato

Para a visualização detalhada das movimentações e do saldo de uma conta, esta tela inicia com um cabeçalho formatado através de `printf()`. Inicialmente, verifica-se a existência de contas registradas (`totalContas`). Caso não haja contas, uma mensagem informativa é exibida, seguida por `getchar()`, `limpar_tela()` e `return` para retornar ao menu principal.

Havendo contas, estas são listadas, exibindo seu ID e Nome por meio de um loop `for`. O usuário é então solicitado a inserir o ID da conta desejada para visualizar o extrato, utilizando `scanf()` para leitura e `limpar_buffer()` para limpar o buffer de entrada.

Um loop `for` busca a conta correspondente ao ID fornecido. Ao encontrar a conta, o sistema solicita ao usuário que digite a data de início e a data final para o período do extrato desejado, utilizando `scanf()` e `limpar_buffer()` para cada entrada.

Uma validação é realizada através da função `compararDatas()` para garantir que a data de início não seja posterior à data final. Em caso positivo, uma mensagem de erro é exibida, seguida por `getchar()`, `limpar_tela()` e `return`.

O sistema então informa o período do extrato que será exibido, utilizando as datas de início e fim fornecidas.

As movimentações da conta são ordenadas por data dentro do período especificado através da função `ordenarMovimentacoesPorData()`.

Um loop `for` itera sobre as movimentações da conta. Para cada movimentação, a função `compararDatas()` é utilizada para verificar se a data da movimentação está dentro do período definido (entre a data de início e a data final, inclusive). Se a movimentação estiver dentro do período, suas informações (Nome, Valor, Data, Entrada/Saída, Tipo) são exibidas formatadamente através de `printf()`.

Durante a exibição das movimentações dentro do período, o saldo parcial (`saldoPeriodo`) é calculado. Para contas de débito (`conta->tipo == 'D'`), o valor da movimentação é adicionado se for uma entrada (`mov->pagamento == 'E'`) é subtraído se for uma saída (`mov->pagamento == 'S'`). Para contas de crédito (`conta->tipo == 'C'`), o valor da movimentação é sempre subtraído.

Após a exibição de todas as movimentações dentro do período, o saldo parcial calculado para o período e o saldo atual da conta (conta->valor) são exibidos através de printf(). O sistema então pausa com getchar(), limpa a tela com limpar_tela() e retorna ao menu principal.

Caso o ID da conta fornecida não seja encontrado após a busca, uma mensagem de erro ("Id não encontrado") é exibida, seguida por getchar() e limpar_tela() antes de retornar ao menu.

```
1 void exibirSaldoExtrato(){
2     printf("=====\n");
3     printf("| Extrato e Saldo da conta |\n");
4     printf("=====\n\n");
5
6     if(totalContas == 0){
7         printf("Nenhuma conta encontrada. Registre uma conta primeiro!!!");
8         getchar();
9         limpar_tela();
10        return;
11    }
12
13    printf("Contas encontradas:\n");
14    for (int i = 0; i < totalContas; i++){
15        printf("\nID - %d | Nome - %s", contas[i].id, contas[i].nome);
16    }
17
18    int buscaId;
19    printf("\n\nDigite o número do ID da conta: ");
20    scanf("%d", &buscaId);
21    limpar_buffer();
22
23
24    for (int i = 0; i < totalContas; i++){
25        if(buscaId == contas[i].id){
26            Conta conta = &contas[i];
27            char inicio[11], fim[11];
28            float saldoPeriodo = 0;
29
30            printf("\n\nDigite a data inicio da procura: ");
31            scanf("%10s", inicio);
32            limpar_buffer();
33            printf("\n\nDigite a data final da procura: ");
34            scanf("%10s", fim);
35            limpar_buffer();
36
37            if(compararDatas(inicio, fim) > 0){
38                printf("\n [ERROR] - A data de inicio é maior que a data final!!!");
39                getchar();
40                limpar_tela();
41                return;
42            }
43
44            printf("\nMovimentações no periodo %s a %s:\n", inicio, fim);
45
46            // Ordena a movimentação, nessa função ela só vai ordenar as movimentações do periodo
47            ordenarMovimentacoesPorData(conta);
48
49            for (int j = 0; j < conta->totalMovimentacoes; j++){
50                Movimentacao* mov = &conta->movimentacoes[j];
51
52                if (compararDatas(inicio, mov->data) <= 0 && compararDatas(fim, mov->data) >= 0){
53                    printf("\n-----");
54                    printf("\nNome da movimentação: %s", mov->nome);
55                    printf("\nValor: %.2f", mov->valor);
56                    printf("\nData: %s", mov->data);
57                    printf("\nEntrada/Saida: %c", mov->pagamento);
58                    printf("\nTipo da movimentação: %c", mov->tipo);
59
60                    // Calcular o saldo parcial com base no tipo da conta e movimentação
61                    if (conta->tipo == 'D') {
62                        if (mov->pagamento == 'E') {
63                            saldoPeriodo += mov->valor;
64                        }
65                        else if (mov->pagamento == 'S'){
66                            saldoPeriodo -= mov->valor;
67                        }
68                    } else if (conta->tipo == 'C') {
69                        saldoPeriodo -= mov->valor;
70                    }
71                }
72                printf("\n=====");
73                printf("\nSaldo no periodo: %.2f", saldoPeriodo);
74                printf("\nSaldo atual da conta é %.2f", conta->valor);
75                getchar();
76                limpar_tela();
77                return;
78            }
79        }
80    }
81    printf("Id não encontrado");
82    getchar();
83    limpar_tela();
84 }
```

Figura 11. Código da tela de exibir o saldo

3.6 Tela de todas as movimentações por conta

Essa funcionalidade exibe todas as movimentações realizadas em uma conta, agrupadas e organizadas por tipo (entrada ou saída), apresentando também o saldo total acumulado. O sistema utiliza algoritmos de ordenação para exibir os dados de forma clara e organizada.

```
=====
|  Todas as movimentações por conta  |
=====

Todas as contas encontradas:

==Nome da conta: teste==

  1 - Nome: testando | Valor: 100,00 | Data: 23/02/2001 | Tipo: E | Transação: E

0 total de movimentações dessa conta foi: 1
==Nome da conta: Bradesco==

  1 - Nome: caderno | Valor: 10,00 | Data: 01/02/2024 | Tipo: E | Transação: S

0 total de movimentações dessa conta foi: 1_
```

Figura 12. Tela de todas as movimentações por conta

Para exibir um panorama completo das movimentações de cada conta registrada, esta tela inicia com um cabeçalho formatado através de `printf()`. Inicialmente, verifica-se a existência de contas (`totalContas`). Caso não haja contas, uma mensagem informativa é exibida, seguida por `getchar()`, `limpar_tela()` e `return` para retornar ao menu principal.

Existindo contas, o sistema informa que todas as contas encontradas serão listadas.

Um loop for itera sobre cada conta registrada no *array* contas. Para cada conta:

1. Um ponteiro Conta *conta é criada para facilitar o acesso aos dados da conta atual.
2. Todas as movimentações associadas a essa conta são ordenadas por data através da função ordenarMovimentacoesPorData(conta).
3. O nome da conta atual (conta->nome) é exibido em um cabeçalho formatado.
4. Verifica-se se há movimentações registradas para essa conta (conta->totalMovimentacoes).
 - Se não houver movimentações, uma mensagem informa: "Não há movimentações registradas para esta conta."
 - Se houver movimentações, outro loop for itera sobre o *array* de movimentações da conta. Para cada movimentação, suas informações (j+1 como um índice de exibição, nome, valor, data, tipo, pagamento) são exibidas formatadamente através de printf(). Após listar todas as movimentações da conta, o número total de movimentações para essa conta (conta->totalMovimentacoes) é exibido.

Após processar e exibir as movimentações de todas as contas, o sistema pausa com getch() e limpa a tela com limpar_tela() antes de retornar ao menu principal.

```

1 void totalMovimentacao(){
2     printf("=====\n");
3     printf("| Todas as movimentações por conta |\n");
4     printf("=====\n\n");
5
6     if(totalContas == 0){
7         printf("Nenhuma conta encontrada. Registre uma conta primeiro!!!");
8         getchar();
9         limpar_tela();
10        return;
11    }
12
13    printf("Todas as contas encontradas:\n");
14
15    for (int i = 0; i<totalContas; i++){
16        Conta *conta = &contas[i];
17
18        // Ordena todas as movimentações de cada conta nessa função
19        ordenarMovimentacoesPorData(conta);
20
21        printf("\nNome da conta: %s==\n", conta->nome);
22        if (conta->totalMovimentacoes == 0) {
23            printf("Não há movimentações registradas para esta conta.\n");
24        }else{
25            for (int j = 0; j<conta->totalMovimentacoes; j++){
26                Movimentacao* mov = &conta->movimentacoes[j];
27                printf("\n %d - Nome: %s | Valor: %.2f | Data: %s | Tipo: %c | Transação: %c", j+1,mov->nome,mov->valor,mov->data,mov->tipo,mov->pagamento);
28            }
29            printf("\n0 total de movimentações dessa conta foi: %d", conta->totalMovimentacoes);
30        }
31    }
32
33    getchar();
34    limpar_tela();
35 }
36

```

Figura 13. Código da tela de total de movimentações

3.7 Tela de Geração de Arquivo CSV

Por meio dessa funcionalidade, o usuário pode exportar os dados financeiros para um arquivo no formato CSV, amplamente compatível com editores de planilha como Microsoft Excel e Google Sheets. Essa exportação permite arquivamento, análise externa e compartilhamento dos dados.

```

=====
| Gerador do arquivo |
=====

Arquivo CSV 'Sistema_de_Contas.csv' gerado com sucesso!

```

Figura 14. Tela de geração de arquivo

Função salvar_dados (Salvar Dados em Binário):

Esta função (salvar_dados) é responsável por persistir os dados do sistema (o próximo id disponível, o número total de contas e o *array* de contas com suas respectivas informações) em um arquivo binário especificado pelo parâmetro arquivo.

1. Abre o arquivo no modo de escrita binária ("wb").
2. Verifique se houve erro na abertura e retorna em caso afirmativo.
3. Escreve o valor da variável global id no arquivo.
4. Escreve o valor da variável global totalContas no arquivo.
5. Escreve todo o conteúdo do *array* global contas no arquivo.
6. Fecha o arquivo.

Função carregar_dados (Carregar Dados em Binário):

Esta função (carregar_dados) tem como objetivo ler os dados do sistema previamente salvos em um arquivo binário especificado pelo parâmetro arquivo e restaurar o estado do sistema.

1. Abre o arquivo no modo de leitura binária ("rb").
2. Se a abertura falhar (arquivo não existe), retorna sem fazer nada.
3. Lê um valor do arquivo e armazena na variável global id.
4. Lê o valor do número total de contas do arquivo e armazena em totalContas.
5. Lê os dados das contas do arquivo e preenche o *array* global contas.
6. Fecha o arquivo.

Função gerarArquivo (Exportar Dados para CSV - Texto):

Esta função (gerarArquivo) tem como propósito exportar os dados do sistema (informações das contas e suas respectivas movimentações) para um arquivo no formato CSV (Comma Separated Values), nomeado "Sistema_de_Contas.csv".

1. Abre (ou cria) o arquivo "Sistema_de_Contas.csv" no modo de escrita de texto ("w").
2. Verifica se houve erro na abertura/criação e retorna em caso afirmativo.
3. Escreve a linha de cabeçalho para a tabela de contas.
4. Itera sobre todas as contas e escreve seus dados no arquivo CSV.

5. Adiciona uma linha em branco para separar as tabelas.
6. Escreve a linha de cabeçalho para a tabela de movimentações.
7. Itera sobre todas as contas e, para cada conta com movimentações, itera sobre suas movimentações e escreve seus dados no arquivo CSV.
8. Fecha o arquivo.
9. Exibe uma mensagem de sucesso para o usuário e pausa a execução.

```

1 // Salvar dados em Binário
2 void salvar_dados(const char *arquivo){
3     FILE *fp = fopen(arquivo, "wb");
4     if (fp == NULL){
5         perror("Erro ao abrir o arquivo para salvar");
6         return;
7     }
8
9     fwrite(&id, sizeof(int), 1, fp);
10    fwrite(&totalContas, sizeof(int), 1, fp);
11    fwrite(contas, sizeof(Conta), totalContas, fp);
12
13    fclose(fp);
14 }
15
16 // Carregar dados em Binário
17 void carregar_dados(const char *arquivo){
18     FILE *fp = fopen(arquivo, "rb");
19     if (fp == NULL){
20         // Arquivo não existe ainda
21         return;
22     }
23
24     fread(&id, sizeof(int), 1, fp);
25     fread(&totalContas, sizeof(int), 1, fp);
26     fread(contas, sizeof(Conta), totalContas, fp);
27
28     fclose(fp);
29 }
30
31 // Exportar dados para CSV(texto)
32 void gerarArquivo(){
33     FILE *fp = fopen("Sistema_de_Contas.csv", "w");
34     if (fp == NULL) {
35         perror("Erro ao criar o arquivo CSV");
36         return;
37     }
38
39     // Cabeçalho da Tabela de Contas
40     fprintf(fp, "ID_Conta;Nome_Conta;Saldo/Crédito\n");
41
42     // Escrever as contas e seus valores
43     for (int i = 0; i < totalContas; i++) {
44         Conta *conta = &contas[i];
45         fprintf(fp, "%d;%s;%.2f\n", conta->id, conta->nome, conta->valor);
46     }
47
48     // Adiciona uma linha em branco para separar as tabelas
49     fprintf(fp, "\n");
50
51     // Cabeçalho da Tabela de Movimentações
52     fprintf(fp, "ID_Conta;Nome_Conta;Tipo_Conta;Nome_Movimentacao;Valor;Data;Tipo;Pagamento\n");
53
54     // Percorrer contas e movimentações
55     for (int i = 0; i < totalContas; i++) {
56         Conta *conta = &contas[i];
57
58         // Se a conta tiver movimentações, percorre as movimentações
59         if (conta->totalMovimentacoes > 0) {
60             for (int j = 0; j < conta->totalMovimentacoes; j++) {
61                 Movimentacao *mov = &conta->movimentacoes[j];
62                 fprintf(fp, "%d;%s;%c;%s;%.2f;%s;%c;%c\n",
63                     conta->id, conta->nome, conta->tipo, mov->nome,
64                     mov->valor, mov->data, mov->tipo, mov->pagamento);
65             }
66         }
67     }
68
69     // Fechar o arquivo
70     fclose(fp);
71
72     printf("=====\n");
73     printf("| Gerador do arquivo | \n");
74     printf("=====\n\n");
75     // Mensagem de sucesso
76     printf("Arquivo CSV 'Sistema_de_Contas.csv' gerado com sucesso!\n");
77     getchar();
78 }
79

```

Figura 15. Código da tela de persistência de dados

4 - Funções auxiliares e algoritmos utilizados

A seguir, examinaremos as funções auxiliares, cruciais para o funcionamento das funcionalidades do código, implementadas nos arquivos `faux.c` e `faux.h`, detalhando os algoritmos utilizados em cada uma delas.

4.1 `limpar_tela()`

Responsável por limpar a saída do console, proporcionando uma interface visual organizada para o usuário. O algoritmo utilizado realiza uma **seleção condicional** baseada no sistema operacional. Se o sistema for Windows, executa o comando `cls`. Caso contrário (Mac ou Linux), executa o comando `clear`.

```
1 void limpar_tela(){
2     #ifdef _WIN32
3         system("cls"); // Se for windows
4     #else
5         system("clear"); // Se for Mac ou Linux
6     #endif
7 }
```

Figura 16. Código de limpar tela

4.2 `limpar_buffer()`

Objetiva limpar o buffer de entrada padrão (`stdin`). O algoritmo implementado utiliza uma **iteração (loop while)** que lê e descarta caracteres do buffer até encontrar o caractere de nova linha (`\n`) ou o fim do arquivo (EOF).

```
1 void limpar_buffer(){
2     int c;
3     while ((c = getchar()) != '\n' && c != EOF); //
        Para limpar o buffer, que sempre trava o
        programa! >:(
4 }
```

Figura 17. Código de limpar o buffer

4.3 CompararDatas(const char *data1, const char *data2)

Realiza a comparação cronológica entre duas datas fornecidas como *strings* no formato "DD/MM/AAAA". O algoritmo segue uma **comparação hierárquica**: primeiro compara os anos, depois os meses e, por fim, os dias. Retorna -1 se data1 é anterior, 1 se posterior e 0 se as datas forem iguais.

```
1  int compararDatas(const char *data1, const char *data2
   ){
2      int d1, m1, a1;
3      int d2, m2, a2;
4
5      // Extrai os componentes de data1
6      sscanf(data1, "%d/%d/%d", &d1, &m1, &a1);
7
8      // Extrai os componentes de data2
9      sscanf(data2, "%d/%d/%d", &d2, &m2, &a2);
10
11     // Primeiro compara pelo ano
12     if (a1 < a2) return -1;
13     if (a1 > a2) return 1;
14
15     // Se os anos forem iguais, compara o mês
16     if (m1 < m2) return -1;
17     if (m1 > m2) return 1;
18
19     // Se os anos e meses forem iguais, compara o dia
20     if (d1 < d2) return -1;
21     if (d1 > d2) return 1;
22
23     // As datas são iguais
24     return 0;
25 }
```

Figura 18. Código para comprar datas

4.4 removerItem(Conta *conta, int indice)

Remove uma movimentação financeira do *array* de movimentações associado a uma conta (Conta). O algoritmo envolve uma **validação de índice** seguida por um **deslocamento de elementos (loop for)**. Os elementos após o índice fornecido são movidos uma posição para a esquerda, sobrescrevendo o item a ser removido. O contador total de movimentações da conta é então decrementado.

```
1  void removerItem(Conta *conta, int indice){
2      if (indice < 0 || indice >= conta
        ->totalMovimentacoes){
3          printf("Índice inválido!\n");
4          return;
5      }
6
7      for (int i = indice; i < conta->totalMovimentacoes
        -1; i++){
8          conta->movimentacoes[i] = conta
        ->movimentacoes[i+1];
9      }
10
11     conta->totalMovimentacoes--;
12 }
13
```

19. Código para remover um item do array

4.5 atualizarSaldoConta(Conta *conta, Movimentacao *mov)

Atualiza o saldo da conta (*conta->valor*) com base no *tipo* da conta e no tipo da movimentação. O algoritmo utiliza **seleção condicional** para determinar a operação a ser realizada: adição para entradas e subtração para saídas em contas de débito, e subtração para todas as movimentações em contas de crédito.

```

1  void atualizarSaldoConta(Conta *conta, Movimentacao
    *mov){
2      if (conta->tipo == 'D') {
3          // Conta de débito
4          if (mov->pagamento == 'E') {
5              conta->valor += mov->valor; // Entrada
6          } else if (mov->pagamento == 'S') {
7              conta->valor -= mov->valor; // Saída
8          }
9      } else if (conta->tipo == 'C') {
10         // Conta de crédito (só sai dinheiro)
11         conta->valor -= mov->valor;
12     }
13 }

```

Figura 20. Código de atualiza o saldo da conta

4.6 reverterSaldoConta(Conta *conta, Movimentacao *mov)

Desfaz a alteração no saldo da conta causada por uma movimentação. O algoritmo emprega **seleção condicional** similar a atualizarSaldoConta(), mas aplica a operação inversa (subtração para entradas revertidas e adição para saídas revertidas em contas de débito; adição para estornos em contas de crédito).

```

1  void reverterSaldoConta(Conta *conta, Movimentacao
    *mov){
2      if (conta->tipo == 'D') {
3          // Conta de débito
4          if (mov->pagamento == 'E') {
5              conta->valor -= mov->valor; // Estornando
                entrada
6          } else if (mov->pagamento == 'S') {
7              conta->valor += mov->valor; // Estornando
                saída
8          }
9      } else if (conta->tipo == 'C') {
10         // Conta de crédito
11         conta->valor += mov->valor; // Estorno em
                crédito
12     }
13 }

```

Figura 21. Código para estornar o saldo

4.7 ordenarMovimentacoesPorData(Conta* conta)

Implementa o algoritmo de ordenação **Bubble Sort** para organizar as movimentações financeiras de uma conta por data. O algoritmo realiza **iterações aninhadas (loops for)** para comparar pares adjacentes de movimentações e **trocar (swap)** suas posições se estiverem fora da ordem cronológica, repetindo o processo até que todas as movimentações estejam ordenadas.

```
1 void ordenarMovimentacoesPorData(Conta* conta){
2     for (int i = 0; i < conta->totalMovimentacoes - 1;
3         i++) {
4         for (int j = 0; j < conta->totalMovimentacoes
5             - i - 1; j++) {
6             // Compara as datas das movimentações
7             if (compararDatas(conta->movimentacoes[j]
8                 .data, conta->movimentacoes[j + 1]
9                 .data) > 0) {
10                // Troca as movimentações se estiverem
11                fora de ordem
12                Movimentacao temp = conta
13                    ->movimentacoes[j];
14                conta->movimentacoes[j] = conta
15                    ->movimentacoes[j + 1];
16                conta->movimentacoes[j + 1] = temp;
17            }
18        }
19    }
20 }
```

Figura 22. Código de ordenação de movimentações

5 - Conclusão

O desenvolvimento deste sistema de gerenciamento de contas representou um desafio multifacetado e enriquecedor para a equipe. A necessidade de aprendizado da linguagem de programação C, com sua sintaxe peculiar e gerenciamento de memória detalhado, demandou um esforço significativo de estudo e adaptação. Paralelamente, a aplicação de princípios de gerenciamento de projetos, desde a concepção até a implementação das funcionalidades, exigiu organização, planejamento e comunicação eficaz entre os membros.

A compreensão e implementação de diversos algoritmos, como o Bubble Sort para ordenação e as lógicas específicas para manipulação de dados nas funções auxiliares, foram cruciais para o funcionamento eficiente do sistema. A escolha de estruturas de dados como *structs* e *arrays* para o armazenamento das informações de contas e movimentações também impôs um aprendizado prático sobre as melhores abordagens para a organização e o acesso aos dados em C.

Apesar dos desafios, o projeto culminou em um sistema funcional que oferece as operações essenciais para o gerenciamento de contas e movimentações. No entanto, reconhecemos o potencial para futuras implementações e aprimoramentos. Algumas áreas que podem ser exploradas incluem:

- **Implementação de algoritmos de ordenação mais eficientes:** Para um grande volume de movimentações, algoritmos como Merge Sort ou Quick Sort poderiam otimizar a performance da ordenação.
- **Aprimoramento da interface do usuário:** A exploração de bibliotecas gráficas ou a criação de uma interface web poderia proporcionar uma experiência de usuário mais intuitiva e visualmente agradável.
- **Implementação de funcionalidades de busca e filtragem avançadas:** Permitir que os usuários busquem movimentações por critérios como nome, valor ou tipo, e filtrem por períodos específicos com maior flexibilidade.
- **Adição de recursos de segurança:** Implementar mecanismos de autenticação e criptografia para proteger os dados dos usuários.

- **Geração de relatórios mais detalhados:** Criar a capacidade de gerar relatórios personalizados sobre o saldo, as movimentações por período, categorias de gastos, etc.
- **Persistência de dados mais robustos:** Explorar o uso de bancos de dados para um gerenciamento de dados mais escalável e confiável.

Em suma, este projeto não apenas nos proporcionou um aprendizado prático e aprofundado em C, gerenciamento de projetos e algoritmos, mas também nos abriu um leque de possibilidades para a evolução contínua deste sistema, visando atender às necessidades dos usuários de forma cada vez mais completa e eficiente.