Universidade Católica de Brasília

Pró-Reitoria Acadêmica
Curso de Engenharia de Software

Banco Malvader

Autor: Guilherme Silvestre Barcelos
Caio Gasparin Siqueira de Andrade
Gabriel Almeida Silva Netto
Arthur Moreira Alves da Silva
Gabriel Reis Oliveira Sousa

Orientador: William Roberto Malvezzi

Brasília - DF
2025

Relatório sobre o Banco

Este relatório documenta a implementação do sistema bancário "Banco Malvader", focando na aplicação prática dos tópicos essenciais de programação em C, conforme demonstrado no código-fonte do projeto.

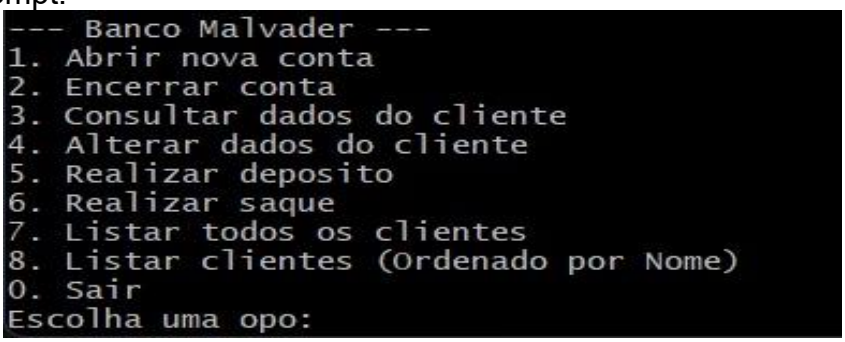
1. Introdução e Arquitetura do Sistema:

O objetivo geral do trabalho foi construir um sistema em C, chamado "Banco Malvader", com a capacidade de gerenciar o cadastro de clientes e realizar operações financeiras básicas. Desse modo, dividimos o código em pastas que se conectam entre si e o desenvolvimento foi focado na aplicação prática de tópicos sólidos de programação em C, como modularização, ponteiros, alocação dinâmica e algoritmos de ordenação.

Os objetivos funcionais são:

- Gerenciamento de Contas:
  - Abertura de novas contas de clientes.
  - Encerramento de contas existentes (exigindo saldo zero).
  - Consulta detalhada e alteração de informações cadastrais do cliente.
- Operações financeiras:
  - Consultar saldo da conta
  - Realizar deposito
  - Processar saques do cliente
  - Registro de movimentos
  - Relatórios do cliente ordenados

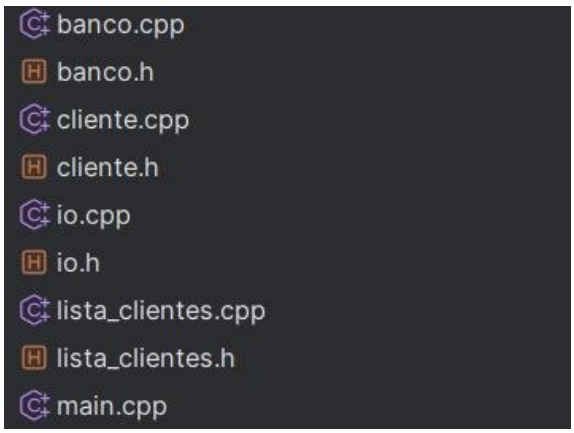
Exemplo do prompt:



Aqui demonstra as escolhas de gerenciamento de conta e das operações financeiras que o cliente pode fazer, logo depois que executa o Banco, através das opções de 1 a 8 e o 0 para finalizar a execução do sistema.

Modularização:

O projeto foi organizado em diferentes módulos, de acordo com o princípio da modularização para a separação das responsabilidades, com o objetivo de facilitar a manutenção e reutilização do código. O sistema foi dividido em camadas logicas distribuídas em .c e .h:



Definição dos módulos:

- main.c
  - Camada da interface, onde tudo se conecta: Exibe menus, coleta entrada do usuário e coordena o fluxo do programa.
- cliente.c / cliente.h
  - Camada de Domínio: Contém a definição da struct Cliente e funções de manipulação de dados em nível de registro.
- banco.c / banco.h
  - Camada de Serviços: Implementa as operações de alto nível sobre a estrutura de controle Banco.
  - Camada de Persistência e Entrada/Saída: Contém rotinas para leitura e escrita segura de dados.
- lista_clientes.c / lista_clientes.h
  - Camada de Estrutura de Dados: Gerencia o vetor dinâmico de clientes. É responsável por encapsular o ponteiro (Cliente* dados), o tamanho e a capacidade (Lista), e funções como inicialização e adição (lista_add)

---

2. Modelagem de Dados e Gerenciamento de Memória:

Essa parte e destinada a demonstrar os tipos de dados complexos do tipo struct e o uso de alocação dinâmica de memória para a construção do código.

2.1 Agrupando Informações com Registros (struct)

Para modelar uma entidade complexa como o cliente bancário, utilizamos o tipo registro (struct). A struct permite agrupar variáveis de tipos diferentes (texto para nome, ponto flutuante para saldo, etc.) sob uma única entidade lógica, o Cliente.

A definição da struct Cliente encapsulou todos os dados cadastrais e financeiros. Para simplificar a sintaxe no código, utilizamos o typedef.

É importante notar que, conforme a regra de negócio do projeto, os dados financeiros essenciais, como saldo e senha, foram incluídos diretamente na struct Cliente, e não em uma estrutura Conta separada. O campo ativo (um int) é usado para marcar o status da conta (1 para ativa, 0 para encerrada).

```c
// Definindo a struct Cliente conforme o modelo
typedef struct {
    char agencia[8];
    char conta[16];
    char nome[200];
    char cpf[14];
    char data_nasc[14];
    char telefone[22];
    char endereco[120];
    char cep[12];
    char local[60];
    char numero_casa[6];
    char bairro[60];
    char cidade[60];
    char estado[4];
    char senha[20];
    double saldo;
    int ativo; // 1 para ativa, 0 para encerrada
} Cliente;
```

2.2 Gerenciando Múltiplos Clientes com Vetores Dinâmicos

Para lidar com um número variável e desconhecido de clientes, implementamos uma coleção baseada em alocação dinâmica de memória, essencial para construir uma solução eficiente e flexível em C.

Estrutura de Controle (Lista / TAD)

Ao invés de manipular o ponteiro para o vetor diretamente, criamos uma estrutura de controle, a struct Lista (representando o banco). Essa estrutura age como um Tipo Abstrato de Dados (TAD) rudimentar, encapsulando a complexidade do gerenciamento da memória e da coleção:

```c
// Estrutura para gerenciar uma lista dinâmica de clientes
typedef struct {
    Cliente* dados; // Ponteiro para o vetor de clientes
    size_t tam;     // Número atual de clientes na lista
    size_t cap;     // Capacidade máxima atual do vetor
} ListaClientes;
```

Controle de Memória e Vetores Dinâmicos

A construção de um conjunto baseado em alocação de memória dinâmica é essencial para o projeto "Banco Malvader", possibilitando lidar com uma quantidade incerta e variável de clientes de maneira eficaz e maleável em C.

2.2.1 Emprego do malloc (Alocação Preliminar)

```c
// Criar uma cópia do vetor de clientes para ordenar
Cliente* copia_clientes = (Cliente*)malloc(b->clientes.tam * sizeof(Cliente));
if (copia_clientes == NULL) {
    fprintf(stderr, "Erro: Falha ao alocar memória para cópia de clientes.\n");
    return;
}
memcpy(copia_clientes, b->clientes.dados, b->clientes.tam * sizeof(Cliente));
```

Para administrar esse conjunto, elaborou-se a struct Lista, que age como um Tipo Abstrato de Dados (TAD) e se responsabiliza por guardar o ponteiro para as informações (Cliente* dados), o tamanho e a capacidade da lista. A função malloc é usada na fase inicial para reservar o bloco de memória inicial fundamental para guardar o primeiro grupo de exemplos da struct Cliente quando o banco é iniciado ou carregado.

2.2.2 Emprego do realloc (Redimensionamento)

```c
int lista_clientes_add(ListaClientes* L, Cliente c) {

    se (L->Tam == L->boné) {
        size_t nova_capacidade = (L->boné == 0) ? TAMANHO_BLOCO_INICIAL : L->boné * 2;
        Cliente* temp = (Cliente*)REALLOC(L->dados, nova_capacidade * tamanho de(Cliente));

        se (temporário == ZERO) {
            fprintf(stderr, "Erro: Falha ao realocar memória para a lista de clientes.\n");
            retornar 0;
        }
        L->dados = temp;
        L->boné = nova_capacidade;
    }

    L->dados[L->Tam] = c;
    L->Tam++;
    retornar 1;
}
```

A precisão do realloc aparece para assegurar a expansibilidade do sistema. O realloc seria a ferramenta utilizada para ampliar a capacidade do vetor dinâmico (Cliente* dados) quando a capacidade atual for alcançada. Isso viabiliza o acréscimo de novos clientes de forma constante, realocando um novo bloco de memória maior e copiando o conteúdo presente, sem demandar que o programador saiba o número exato de clientes de antemão. (Observação: O relatório aponta a falta da citação clara a malloc e realloc na seção inicial, mostrando que esses métodos de alocação são obrigatórios para o funcionamento da estrutura Lista).

3. Funções, Ponteiros e Segurança

Esta parte e destinada para detalhar o uso correto de ponteiros e a passagem de parâmetros por referência para garantir que as funções possam alterar o estado de objetos externos, além de abordar as técnicas de entrada segura de dados e manipulação de strings exigidas pelo projeto.

3.1 Passagem por Referência e Uso de Ponteiros

O sistema faz uso intensivo de ponteiros para simular a passagem de parâmetros por referência. Isso é essencial para que as funções possam alterar dados de instâncias da struct Cliente alocadas externamente.

- Manipulação de Dados (Saldo): Funções de negócio, como banco_depositar e banco_sacar, recebem um ponteiro para a struct Banco (Banco* b). A partir desse ponteiro, o índice do cliente é localizado, e seu saldo é modificado diretamente usando o operador seta (->), demonstrando a manipulação segura de dados externos.

```c
int banco_depositar(Banco* b, const char* conta, double valor) {
    if (valor <= 0) {
        fprintf(stderr, "Erro: Valor de depósito deve ser positivo.\n");
        return 0;
    }

    int idx = lista_clientes_buscar_por_conta(&b->clientes, conta);
    if (idx == -1) {
        fprintf(stderr, "Erro: Conta %s não encontrada.\n", conta);
        return 0;
    }
    if (!b->clientes.dados[idx].ativo) {
        fprintf(stderr, "Erro: Conta %s está inativa e não pode receber depósitos.\n", conta);
        return 0;
    }

    b->clientes.dados[idx].saldo += valor;
    printf("Depósito de %.2f realizado na conta %s. Novo saldo: %.2f\n", valor, conta, b->clientes.dados[idx]
    registrar_movimento(b->arquivo_movimentos, conta, "DEPOSITO", valor, b->clientes.dados[idx].saldo);
    return 1;
}
```

Manipulação de Elementos (Quick Sort): Para a rotina de ordenação, a função utilitária trocar_clientes é implementada recebendo dois ponteiros para Cliente. Ela desreferencia os ponteiros (*a e *b) para trocar o conteúdo das estruturas na memória, permitindo que a ordenação ocorra in place (no local original).

```c
// Troca dois clientes de posição na memória
void trocar_clientes(Cliente* a, Cliente* b) {
    Cliente temp = *a;
    *a = *b;
    *b = temp;
}
```

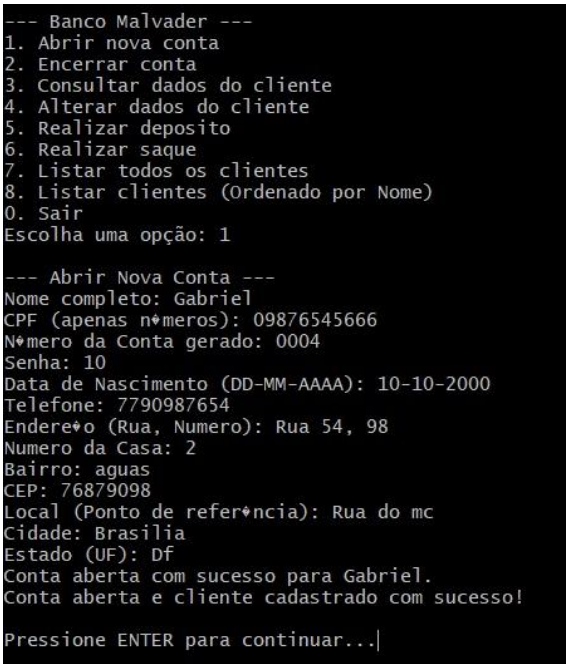
3.2 Strings em C e Entrada Segura de Dados

O projeto prioriza a segurança na manipulação de texto, evitando vulnerabilidades comuns em C, como o *buffer overflow*.

- Entrada Segura (fgets e strcspn): Toda a coleta de entrada de texto (nome, CPF, senha, endereço) é centralizada na função ler_linha. Esta função é implementada conforme a prática sólida de utilizar fgets para limitar o tamanho da leitura e, subsequentemente, strcspn para remover o caractere de nova linha (\n) que o fgets insere, garantindo que a *string* armazenada seja limpa e termine corretamente com \0.

```c
// Lê uma linha de texto de forma segura, removendo o '\n' se presente
void ler_linha(char* buffer, size_t tamanho_maximo) {
    if (fgets(buffer, (int)tamanho_maximo, stdin)) {
        // Remove o '\n' se ele foi lido
        buffer[strcspn(buffer, "\n")] = 0;
    }
}
```

Execução da abertura de conta pelo prompt:



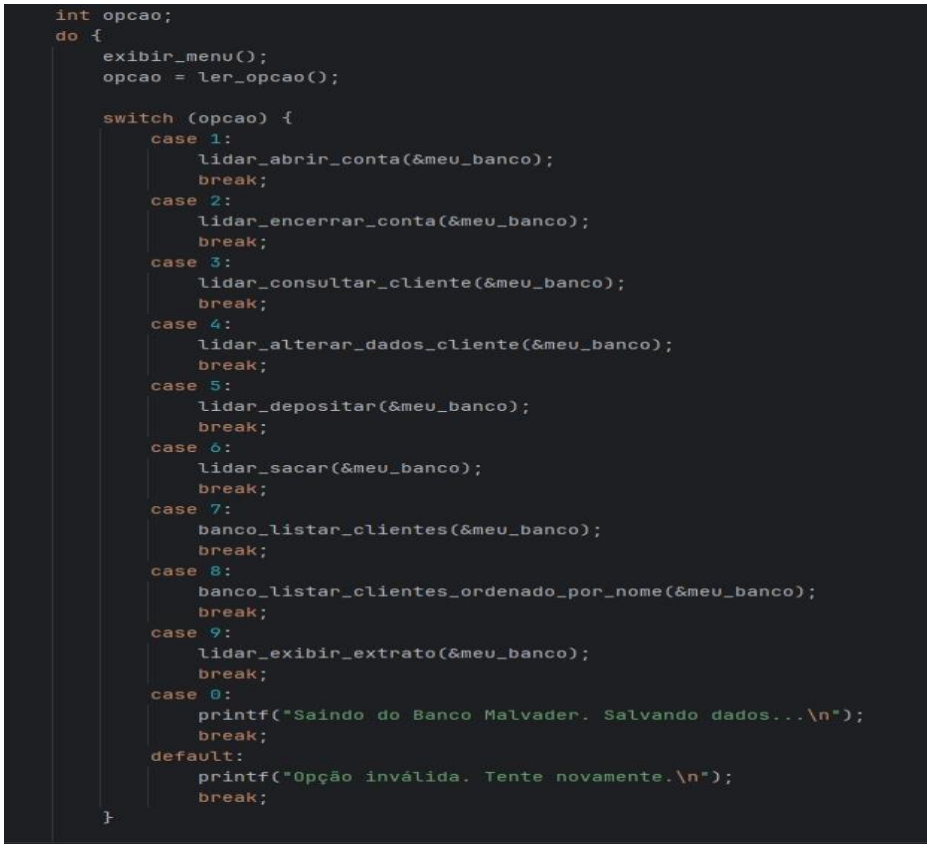
Manipulação de Strings: A comparação de strings (strcmp) é usada extensivamente em funções de busca (por CPF e conta) e nas rotinas de ordenação (por nome).

3.3 Robustez e Validação com do-while

A robustez de interface com o usuário foi garantida através do uso de laços de repetição, com destaque para o do-while.

Menu de Navegação:

O laço do-while é a estrutura principal no arquivo main.c, garantindo que seja exibido o menu e o programa execute as operações pelo menos uma vez antes de verificar a condição de saída (opção 0 - Sair).



```c
int opcao;
do {
    exibir_menu();
    opcao = ler_opcao();

    switch (opcao) {
        case 1:
            lidar_abrir_conta(&meu_banco);
            break;
        case 2:
            lidar_encerrar_conta(&meu_banco);
            break;
        case 3:
            lidar_consultar_cliente(&meu_banco);
            break;
        case 4:
            lidar_alterar_dados_cliente(&meu_banco);
            break;
        case 5:
            lidar_depositar(&meu_banco);
            break;
        case 6:
            lidar_sacar(&meu_banco);
            break;
        case 7:
            banco_listar_clientes(&meu_banco);
            break;
        case 8:
            banco_listar_clientes_ordenado_por_nome(&meu_banco);
            break;
        case 9:
            lidar_exibir_extrato(&meu_banco);
            break;
        case 0:
            printf("Saindo do Banco Malvader. Salvando dados...\n");
            break;
        default:
            printf("Opção inválida. Tente novamente.\n");
            break;
    }
```

- Validação: Embora o do-while do main controle o fluxo, a lógica de validação de dados em outras partes do sistema, como em banco_depositar e banco_sacar, usa verificações e retornos de erro (return 0;), garantindo que o estado do banco só seja alterado se as regras (e.g., valor positivo, saldo suficiente) forem atendidas.

---

4. Algoritmos e Estruturas de Controle

Esta parte detalha a aplicação de um algoritmo de ordenação complexo e o uso da recursividade para manipulação da lista de clientes, além de garantir o controle de fluxo do programa.

4.1 Ordenação com Quick Sort e Recursividade

A listagem de clientes ordenada por nome (opção 8 do menu) é implementada por meio de uma versão própria do algoritmo Quick Sort, demonstrando o domínio da recursividade.

- Lógica Recursiva: A função quicksort_clientes_por_nome é o coração da ordenação. Ela chama a si mesma duas vezes após a partição (quicksort_particao_nome),

utilizando a recursão para ordenar as sub-listas de elementos. A condição de parada recursiva é definida por if (l < r).

```
void quicksort_clientes_por_nome(Cliente v[], int l, int r) {
    if (l < r) {
        int pi = quicksort_particao_nome(v, l, r);
        quicksort_clientes_por_nome(v, l, pi - 1);
        quicksort_clientes_por_nome(v, pi + 1, r);
    }
}
```

- Implementação de Particionamento: A função quicksort_particao_nome usa strcmp para comparar os campos nome das structs Cliente e invoca trocar_clientes (que recebe ponteiros) para mover os elementos, garantindo que a ordenação seja feita in-place.

- Gerenciamento de Memória para Ordenação: Antes de ordenar, a função banco_listar_clientes_ordenado_por_nome cria uma cópia profunda do vetor de clientes usando malloc e memcpy. Isso garante que a ordenação não afete a ordem original da lista em memória, conforme o princípio de imutabilidade de dados.

Listagem de clientes em execução no prompt:

```
--- Lista de Clientes (Ordenada por Nome) ---
--- Dados do Cliente ---
Agncia: 0001
Conta: 0002
Nome: gab
CPF: 175.845.476.8
Data Nasc.: 10-11-2006
Telefone: 61-985963447
Endereo: ,  - ,  - /
CEP: 72015-035
Saldo: 0.00
Status: Ativa
------------------------
--- Dados do Cliente ---
Agncia: 0001
Conta: 0001
Nome: gui
CPF: 175
Data Nasc.: 10-11-2006
Telefone: 61985963447
Endereo: ,  - ,  - /
CEP:
Saldo: 0.00
Status: Ativa
------------------------
--- Dados do Cliente ---
Agncia: 0001
Conta: 0003
Nome: gui
CPF: 17584547680
Data Nasc.: 10-11-2006
Telefone: 61-985963447
Endereo: rua o, 12 - centro, bk - df/df
CEP: 72015-035
Saldo: 0.00
Status: Ativa
------------------------
```

---

5. Persistência de Dados e Arquivos

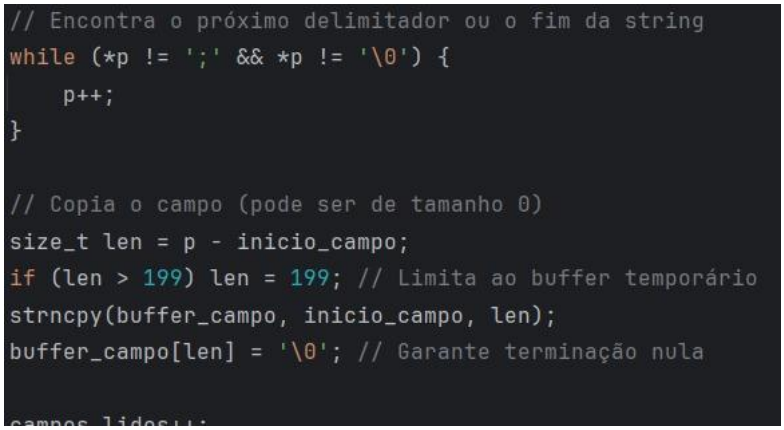
A persistência foi implementada para que o estado do sistema (clientes.txt) e o histórico de transações (movimentos.txt) sobrevivam após o encerramento do programa, utilizando o formato de arquivos texto.

5.1 Salvamento e Carregamento do Estado (clientes.txt)

O arquivo clientes.txt armazena o estado completo de todas as contas, incluindo os dados cadastrais e o saldo (%.2f) e o status de ativação (%d).

- Salvamento Confiável: A função salvar_clientes abre o arquivo no modo escrita ("w") e o reescreve completamente (consistência total) usando fprintf.

- Carregamento Robusto: A função carregar_clientes utiliza fgets para ler o arquivo linha por linha. O *parsing* dos 16 campos é feito manualmente, usando manipulação de ponteiros (char* p = linha;) e strncpy para extrair cada campo delimitado por ponto e vírgula. Esse método substitui o sscanf complexo, tornando a leitura mais robusta a erros de formato e tamanho de *buffer*.



```
// Encontra o próximo delimitador ou o fim da string
while (*p != ';' && *p != '\0') {
    p++;
}

// Copia o campo (pode ser de tamanho 0)
size_t len = p - inicio_campo;
if (len > 199) len = 199; // Limita ao buffer temporário
strncpy(buffer_campo, inicio_campo, len);
buffer_campo[len] = '\0'; // Garante terminação nula

campos_lidos++;
```

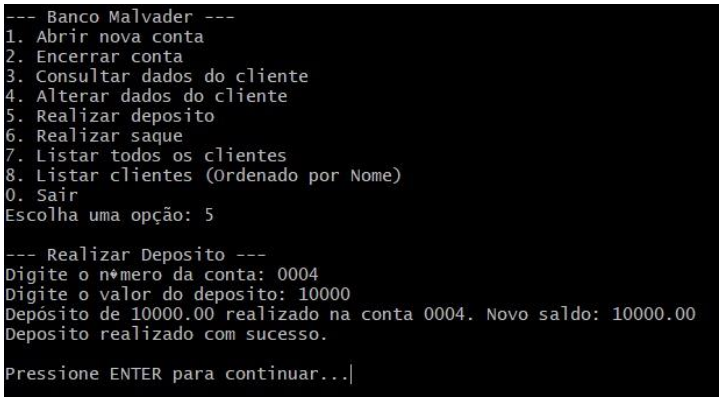
5.2 Registro de Movimentos (movimentos.txt)

O extrato da conta é mantido em um arquivo de log que registra todas as transações, garantindo a rastreabilidade.

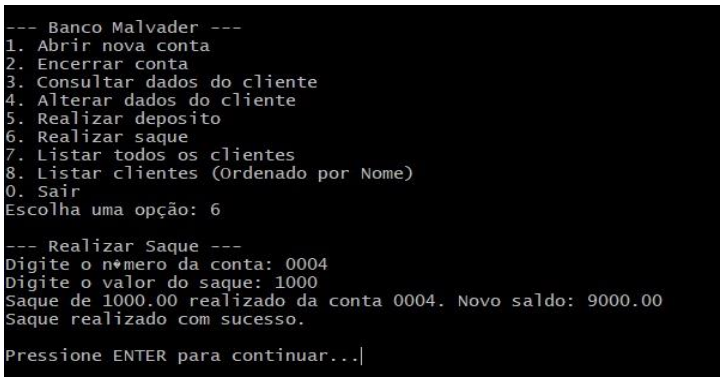
- Modo de Apêndice: A função registrar_movimento abre o arquivo no modo apêndice ("a"), garantindo que cada novo registro (depósito ou saque) seja adicionado ao final do arquivo, preservando o histórico.

- Data Automática: O registro inclui a data da operação, obtida usando funções da biblioteca time.h (time(NULL), localtime), formatada por fprintf.

Operação de deposito:



Operação de saque:



---

6. Conclusão

O projeto "Banco Malvader" alcançou com sucesso todos os objetivos propostos. A construção do sistema demonstrou o domínio de conceitos essenciais da linguagem C, resultando em um código modular, robusto e eficiente:

- Organização e Abstração: A arquitetura em módulos e a criação de um TAD (ListaClientes em lista_clientes.h) garantiram a organização e a abstração da lógica central.

- Gerenciamento de Memória: O uso do realloc seguro no vetor dinâmico e o uso correto de free eliminam *memory leaks* e garantem a flexibilidade da aplicação.
- Ponteiros: A passagem por referência foi aplicada com sucesso em todas as transações e nas rotinas de ordenação, permitindo a modificação do estado do cliente de forma eficiente.

- Algoritmos e Segurança: A implementação do Quick Sort recursivo e a adoção do fgets com strcspn para entrada de dados segura comprovam a aplicação das práticas de programação exigidas.

O sistema final opera de forma funcional e segura, cumprindo os requisitos de simulação bancária.