Relatório Técnico - Unidade 2

Disciplina: Introdução às Técnicas de Programação
Projeto: Duarte Cadastro de Alunos
Aluno: Felipe Augusto de Lima Duarte
Matrícula: 20250025417

1. Introdução e Contexto

O projeto implementa um Sistema CRUD de Gerenciamento Escolar em linguagem C. O objetivo é permitir o Cadastro, Consulta, Atualização e Exclusão de registros de alunos, utilizando uma interface de linha de comando (CLI).

O projeto resolve o problema do gerenciamento básico de cadastros de alunos, controlando de forma segura os dados (nome, idade, matrícula, série) e permitindo que os registros desativados sejam preservados no sistema, garantindo o histórico e a possibilidade de reativação imediata.

A escolha desse projeto se justifica porque o desenvolvimento de um sistema CRUD é uma forma eficaz de englobar os conceitos fundamentais da Unidade 1 (Funções, Vetores, Condicionais e Laços de Repetição). Por já ter trabalho em um projeto WEB tive contato com o CRUD e pude desenvolver e aprender mais ainda sobre o tema, apesar de não utilizar um sistema de armazenamento persistente como um banco de dados, impossibilitando que os cadastros fiquem guardados mesmo após fechar o terminal, tentei simular esse armazenamento utilizando vetores e tive um resultado positivo.

2. Análise Técnica

2.1 Metodologia e Ferramentas

- **Compilador:** MinGW - GCC
- **Editor:** VSCode
- **Execução:** Terminal do VSCode

2.2 Conceitos Aplicados (Unidade 2)

Nesta unidade, o projeto evoluiu com a incorporação de estruturas de dados mais complexas e gerenciamento eficiente de memória, permitindo funcionalidades avançadas como o sistema de notas e a busca textual.

Strings

O tratamento de cadeias de caracteres foi aprofundado para além do armazenamento simples, permitindo operações de busca e segurança na entrada de dados:

- **Segurança na Leitura (`fgets` e `strcspn`):** Em substituição parcial ao `scanf` para textos, o uso de `fgets` permitiu a leitura segura de nomes compostos (com espaços), enquanto `strcspn` foi utilizado para remover o caractere de nova linha (`\n`) residual, garantindo a formatação correta das strings na memória.
- **Busca por Substring (`strstr`):** Implementada na função `busca_textual`, a função `strstr` da biblioteca `<string.h>` permitiu verificar se um termo de busca (nome ou matrícula) está contido dentro dos registros dos alunos, retornando um ponteiro não nulo em caso de sucesso.
- **Cópia de Strings (`strcpy`):** Fundamental para a manipulação dos nomes alocados dinamicamente e para a atribuição das matérias padrão na inicialização do cadastro.

Estruturas de Repetições Aninhadas

A complexidade dos dados exigiu o uso de laços dentro de laços (`for` dentro de `for`) para manipular estruturas multidimensionais:

- **Inicialização da Matriz de Notas:** Na função `cadastro_aluno`, laços aninhados percorrem as 5 matérias (laço externo) e os 3 trimestres (laço

interno) para definir o valor inicial `-1` em todas as posições da matriz, indicando ausência de nota.
- **Exibição e Manipulação de Notas:** Nas funções `consultar_notas` e `alterar_deletar_notas`, a repetição aninhada é utilizada para iterar sobre a matriz `notas[5][3]`, permitindo acessar, exibir ou modificar cada nota individualmente de forma organizada.

Matrizes

Para suportar o sistema escolar, foi introduzida uma matriz bidimensional na estrutura do aluno:

- **Matriz de Notas (`int notas[5][3]`):** Estrutura que armazena as avaliações do aluno. A primeira dimensão (linhas) representa as 5 matérias fixas (Matemática, Português, etc.), e a segunda dimensão (colunas) representa os 3 trimestres do ano letivo.
- **Vetor de Strings (Matérias):** Embora tecnicamente um vetor de strings, a estrutura `char materias_padrao[5][15]` atua como uma matriz de caracteres para armazenar os nomes das disciplinas, sendo acessada via índice para exibir o nome da matéria correspondente à nota.

Ponteiros

O uso de ponteiros foi ampliado para permitir a modificação direta de variáveis fora do escopo local e o gerenciamento de memória:

- **Manipulação de Variáveis de Controle:** A variável `total_alunos` é passada por referência (`int *total_alunos`) para a função `cadastro_aluno`. Isso permite que o incremento no contador de alunos reflita diretamente na variável original na `main`, mantendo a integridade da contagem.
- **Alocação Dinâmica e Arrays:** Na implementação da alocação dinâmica do nome, ponteiros de char (`char *nome`) substituíram os vetores fixos na `struct`, permitindo que o tamanho da string seja definido em tempo de execução.

Alocação Dinâmica

Para otimizar o uso de recursos, o sistema passou a alocar memória sob demanda para os dados textuais:

- **Alocação do Nome (`malloc`):** Na função `cadastro_aluno`, a memória para o nome do aluno não é mais fixa em 50 bytes. O sistema lê o nome em um buffer temporário, calcula o tamanho exato da string (`strlen`) e utiliza

`malloc` para reservar apenas a quantidade necessária de bytes na memória *heap*.

- **Realocação na Atualização (`free` e `malloc`):** Na função `alterar_aluno`, ao modificar um nome, o sistema utiliza `free` para liberar a memória do nome antigo antes de alocar um novo bloco para o novo nome, prevenindo vazamentos de memória (*memory leaks*) e garantindo o uso eficiente da RAM.

2.3 Estrutura de Dados

- **Struct Aluno Refatorada:** A estrutura foi atualizada para incluir o ponteiro `char *nome` (para alocação dinâmica) e a matriz `int notas[5][3]` para o gerenciamento acadêmico.
- **Matriz de Inteiros:** Essencial para tabular as notas relacionando matérias e trimestres.
- **Ponteiros:** Utilizados tanto para referência de variáveis de controle quanto para o gerenciamento dos endereços de memória das strings dinâmicas.

3. Implementação e Reflexão

3.1 Dificuldades Encontradas

- A maior dificuldade dessa unidade certamente foi utilizar ponteiros. É um conceito abstrato inicialmente, que exige ponderação sobre o uso nas primeiras utilizações. Porém, é um conceito muito importante na linguagem C e nas linguagens de programação em geral.

- Outro ponto importante, que inicialmente também deu trabalho, foi a otimização e refatoração do código. O tópico é desafiador pelo caráter unicidade, existem boas práticas a serem seguidas, porém cada código tem suas particularidades, suas funções, e é necessário pensar e analisar cuidadosamente cada mudança realizada para melhorar o desempenho e a legibilidade do código. Durante o processo quebrei ele muitas vezes, bati bastante cabeça, mas foi extremamente importante para melhorar minhas habilidades de programação, que vão além de escrever código, mas que agora se estendem a entender melhor como as estruturas funcionam, além de desenvolver um pensamento crítico e analítico sobre o que eu estou escrevendo.

3.2 Soluções Implementadas

- Me aprofundei melhor no conceito e utilização de ponteiros, a prática foi uma boa amiga nesse processo de aprendizado. Apesar de ainda existirem algumas lacunas pude melhorar minhas habilidades com ponteiros.

- A análise de código para refatoração e otimização certamente é um processo que só a prática pode aperfeiçoar, por vezes, demora, é uma atividade para quebrar a cabeça. Porém, é extremamente proveitosa, analisar e ponderar sobre o próprio código é de grande valia para o autoaprendizado, por vezes sendo mais efetiva na consolidação do conhecimento do que na prática de escrever o código. Tirei bastante proveito desse tempo e pude aprender bastante sobre programação.

3.3 Organização do Código

A organização do projeto priorizou a clareza e a facilidade de manutenção, dividindo as responsabilidades do sistema em funções distintas para cada operação do CRUD. Recentemente, a estrutura do código foi aprimorada com a implementação de funções utilitárias, responsáveis por centralizar validações repetitivas e leituras de índices.

Essa refatoração reduziu drasticamente a duplicação de código e facilitou a inserção de conceitos estudados na segunda unidade, como a alocação dinâmica de memória para o armazenamento otimizado de strings. Atualmente, o código opera de forma coesa e segura.

3.4 Conclusão

A segunda unidade foi essencial para consolidar o aprendizado de estruturas complexas, como matrizes, ponteiros e alocação dinâmica, permitindo que eu transformasse um cadastro simples em um sistema eficiente e otimizado. A experiência de refatorar o código para implementar novas funcionalidades e gerenciar a memória manualmente me mostrou que programar vai muito além de fazer o código rodar, trata-se de criar soluções organizadas e o mais simples possível. O formato de avaliação por projeto continua sendo um grande diferencial, pois os desafios práticos de manipulação de dados e limpeza de código fixaram os conteúdos de forma muito mais sólida do que uma prova teórica faria, aumentando minha segurança para desenvolver minhas habilidades de programação.

3.5 Possíveis Melhorias

- **Modularização:** Acredito que o passo seguinte a ser tomado para continuar desenvolvendo o código é iniciar a modularização dele. A medida que novas funcionalidades são adicionadas o número de funções só aumenta, em determinado ponto a otimização, refatoração e manutenção será dificultosa devido a todo o programa estar inserido em um único arquivo.

- **Log de atividades:** Seria interessante adicionar futuramente um sistema de log de atividades, com a adição de um sistema de notas e melhorias na alteração de um cadastro surge a necessidade de ter um controle sobre o que está sendo feito durante o uso do programa. Armazenar dados como quem alterou o que, quando alterou e como alterou é bastante importante para segurança e escalabilidade de um bom software.

Perguntas Orientadoras

Quais conceitos da Unidade 1 foram aplicados e onde?

- **Strings:** O uso de strings foi aprofundado para além do armazenamento simples. Nesta etapa, foram aplicadas funções da biblioteca <string.h> para manipulação avançada, como o uso de strcpy para cópia de dados em memória dinâmica e, principalmente, o uso de strstr na nova função de busca textual, permitindo verificar se um termo específico está contido no nome ou na matrícula do aluno.
- **Estruturas de Repetição Aninhada:** Esta estrutura tornou-se essencial para manipular matrizes. Ela está presente nas funções de cadastro (para inicializar notas) e nas funções de consulta e alteração de notas, onde um laço for externo percorre as disciplinas e um laço interno percorre os trimestres, permitindo acessar sequencialmente todas as posições da grade de notas.
- **Matrizes:** As matrizes foram introduzidas para organizar os dados acadêmicos de forma tabular. A estrutura int notas[5][3] dentro da struct do aluno cria uma relação direta entre as 5 matérias fixas e os 3 trimestres do ano letivo, permitindo armazenar e gerenciar o desempenho escolar de forma estruturada.
- **Ponteiros:** Os ponteiros assumiram um papel central no gerenciamento de endereços de memória. Eles são utilizados para passar variáveis de controle por referência (como o contador total_alunos) garantindo a persistência das alterações, e fundamentalmente na definição do campo char *nome na struct, permitindo que o programa manipule endereços de memória variáveis para os nomes.
- **Alocação Dinâmica:** A alocação dinâmica foi implementada para otimizar o uso da memória RAM. Nas funções de cadastro e alteração, o código utiliza malloc para reservar apenas a quantidade exata de bytes necessária para armazenar o nome digitado pelo usuário, e free para liberar memória obsoleta durante a atualização de cadastros, evitando desperdícios e vazamentos de memória.

Como a organização em funções facilita a manutenção do código? A estruturação do código em funções demonstrou-se fundamental para a eficiência da manutenção do projeto. Essa organização permitiu testar e depurar cada funcionalidade de forma isolada, garantindo que a correção de um erro em um módulo específico não causasse problemas no restante do programa. Além disso, a criação de funções auxiliares, como verificar_cadastros e obter_indice_aluno, eliminou a redundância de lógicas repetitivas. Isso centraliza o funcionamento do código: caso seja necessário alterar uma regra de validação, a modificação é feita

em um único local e replicada automaticamente para todo o sistema, tornando o processo de atualização muito mais ágil e consistente.

Quais foram os principais desafios técnicos enfrentados? O maior desafio desta unidade foi o uso de ponteiros, um conceito inicialmente abstrato que exige bastante atenção, mas que é essencial para a programação em C. Outro ponto que exigiu esforço foi a otimização e refatoração do código, pois cada mudança precisava ser analisada com cuidado para melhorar o sistema sem causar novos erros. Apesar das dificuldades encontradas durante esse processo, a experiência foi fundamental para minha evolução técnica, ajudando-me a compreender melhor como as estruturas funcionam e a desenvolver um pensamento mais crítico e analítico sobre o código que escrevo.

Como foram implementadas as estruturas de dados complexas (matrizes)? As matrizes foram implementadas dentro da estrutura do aluno para organizar as notas de forma tabular. Utilizei uma matriz de inteiros, onde as linhas representam as 5 matérias fixas e as colunas representam os 3 trimestres do ano letivo. Isso permite que o sistema acesse e armazene cada nota cruzando diretamente a disciplina com o período correto, facilitando a consulta e a organização dos dados.

Qual a estratégia para gerenciamento de memória? Para gerenciar a memória, o sistema usa uma lista fixa para os cadastros dos alunos, mas trata os nomes de forma inteligente e flexível. Ao invés de reservar um espaço grande igual para todos, o programa pede ao computador apenas a quantidade exata de memória necessária para guardar cada nome. Quando um nome precisa ser alterado, o sistema apaga o espaço antigo e reserva um novo sob medida, garantindo que o computador não desperdice memória com espaços vazios ou dados que não são mais úteis.

Como você garante que não há vazamentos de memória? A garantia contra vazamentos de memória está implementada na lógica de atualização dos cadastros. Sempre que é necessário alterar o nome de um aluno, o código executa obrigatoriamente o comando free no ponteiro antigo antes de alocar o novo espaço. Essa sequência lógica impede que blocos de memória fiquem perdidos ou ocupados inutilmente após a modificação dos dados, mantendo o sistema limpo.

Quais vantagens a alocação dinâmica trouxe para seu projeto? A principal vantagem foi a economia e a eficiência no uso da memória RAM. Ao invés de reservar um espaço fixo e grande para todos os nomes (o que desperdiçaria memória com nomes curtos), o sistema agora aloca apenas a quantidade exata de bytes que cada nome precisa. Isso torna o programa mais leve e adaptável, pois consome recursos proporcionalmente à quantidade de dados inseridos.

Como os ponteiros foram utilizados para melhorar a eficiência? Os ponteiros foram essenciais para permitir a manipulação direta dos dados na memória sem

precisar fazer cópias desnecessárias. Eles foram usados para atualizar as variáveis de controle (como o total de alunos) diretamente na função principal e para gerenciar os endereços dos nomes alocados dinamicamente. Isso tornou o tráfego de dados entre as funções mais rápido e permitiu que o código manipulasse estruturas complexas de forma mais ágil.