CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA

FACULDADE DE TECNOLOGIA DE LINS
PROF. ANTÔNIO SEABRA CURSO SUPERIOR DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

Rhodrigo Danniel da Silva Borges Santos

PORTFÓLIO DE PROJETOS ACADÊMICOS

LINS/SP
2º SEMESTRE/2025

Borges, Rhodrigo Danniel da Silva

B732 PORTFÓLIO DE PROJETOS ACADÊMICOS / Rhodrigo Danniel da Silva Borges. — Lins, 2025.

85f.

Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) — Faculdade de Tecnologia de Lins Professor Antonio Seabra: Lins, 2025.

Orientador(a): Dr. Alciano Genovez

1. Linguagem de Programação. 2. Estrutura de Dados. 3. Programação Orientada a Objetos Avançada. 4. Redes de Computadores. 5. Tópicos Especiais em Informática. I. Genovez , Alciano . II. Faculdade de Tecnologia de Lins Professor Antonio Seabra. III. Título.

CDD 004.21

Gerada automaticamente pelo módulo web de ficha catalografica da FATEC Lins mediante dados fornecidos pelo(a) autor(a).

CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA

FACULDADE DE TECNOLOGIA DE LINS
PROF. ANTÔNIO SEABRA CURSO SUPERIOR DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

RHODRIGO DANNIEL DA SILVA BORGES SANTOS

PORTFÓLIO DE PROJETOS ACADÊMICOS

Trabalho de Conclusão de Curso (TCC) apresentado à Faculdade de Tecnologia de Lins para obtenção do Título de Tecnólogo (a) em Análise e Desenvolvimento de Sistemas

Orientador: Prof. Dr. Alciano Gustavo Genovez de Oliveira.

LINS/SP
2º SEMESTRE/2025

RHODRIGO DANNIEL DA SILVA BORGES SANTOS

PORTFÓLIO DE PROJETOS ACADÊMICOS

Trabalho de Conclusão de Curso apresentado à Faculdade de Tecnologia de Lins Prof. Antônio Seabra, como parte dos requisitos necessários para a obtenção do Título de Tecnólogo (a) em Análise e Desenvolvimento de Sistemas sob orientação do Prof. Dr. Alciano Gustavo Genovez de Oliveira.

Data de aprovação: __/__/____

Orientador: Prof. Dr. Alciano Gustavo Genovez de Oliveira

Examinador 1: Prof. Dr. Anderson Pazin

Examinador 2: Prof. Thiago Galdim Bergamaschi

DEDICATÓRIA

A todos que, direta ou indiretamente, contribuíram para a realização deste sonho. Em especial, à minha família, por ser meu porto seguro.

Rhodrigo Danniel da Silva Borges Santos

AGRADECIMENTOS

Em primeiro lugar, a Deus, por me conceder força, saúde e determinação para chegar até a conclusão desta jornada. Aos professores do curso, meu sincero reconhecimento e gratidão por toda a dedicação e compromisso em transmitir seus valiosos conhecimentos, que foram pilares essenciais para a minha formação profissional e pessoal.

Em especial, estendo meu profundo agradecimento ao meu orientador, Prof. Dr. Alciano Gustavo Genovez de Oliveira, por sua paciência inestimável e pelo suporte constante e orientação especializada durante todas as etapas de desenvolvimento deste trabalho.

À minha família, deixo registrada a mais profunda gratidão por todo o amor incondicional, apoio inabalável e compreensão. O incentivo e a crença que depositaram em mim foram a minha maior fonte de motivação, especialmente nos momentos mais desafiadores. Aos amigos e colegas de curso, agradeço pela amizade, pelas palavras de encorajamento e pelo companheirismo que tornaram essa jornada acadêmica mais leve e significativa.

RESUMO

Este trabalho de conclusão de curso foi estruturado como um portfólio de projetos detalhados. Aborda a trajetória de aprendizado prático e teórico nas disciplinas fundamentais do curso de Análise e Desenvolvimento de Sistemas (ADS), como Linguagem de Programação, Estrutura de Dados, Programação Orientada a Objetos Avançada, Redes de Computadores e Tópicos Especiais em Informática. O objetivo é apresentar o desenvolvimento de sistemas de software que aplicam integralmente a teoria aprendida, com uma atenção especial na análise de requisitos, na modelagem de dados e na entrega de uma solução funcional e robusta. A metodologia empregada consiste na documentação formal de cada projeto, destacando as decisões técnicas tomadas, os desafios enfrentados e as soluções implementadas com clareza. Os resultados demonstram a proficiência na implementação de funcionalidades essenciais, como a persistência de dados, o controle de acesso e a conectividade em rede, confirmando a capacidade do aluno de atuar em todas as fases do ciclo de vida de desenvolvimento de um sistema complexo e moderno.

Palavras-Chave: Linguagem de Programação. Estrutura de Dados. Programação Orientada a Objetos Avançada. Redes de Computadores. Tópicos Especiais em Informática.

ABSTRACT

This final course project is structured as a portfolio of detailed projects. It addresses the practical and theoretical learning trajectory in the fundamental disciplines of the Systems Analysis and Development (ADS) course, such as Programming Language, Data Structures, Advanced Object-Oriented Programming, Computer Networks, and Special Topics in Informatics. The objective is to present the development of software systems that fully apply the learned theory, with special attention to requirements analysis, data modeling, and the delivery of a functional and robust solution. The methodology employed consists of the formal documentation of each project, highlighting the technical decisions made, the challenges faced, and the solutions implemented with clarity. The results demonstrate proficiency in implementing essential functionalities, such as data persistence, access control, and network connectivity, confirming the student's ability to work in all phases of the development lifecycle of a complex and modern system.

Keywords: Programming Language, Data Structures, Advanced Object-Oriented Programming, Computer Networks, Informatics Special Topics, Systems Analysis.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ADS	– Análise e Desenvolvimento de Sistemas
C#	– C Sharp
C++	– Cee Plus Plus
CSS	– Cascading Style Sheets
CMD	– Prompt de Comando
DHCP	– Dynamic Host Configuration Protocol
DNS	– Sistema de Nomes de Domínio
EAP	– Estrutura Analítica do Projeto
HTML	– HyperText Markup Language
IDE	– Integrated Development Environment
LAN	– Local Area Network
MAN	– Metropolitan Area Network
MER	– Modelo Entidade-Relacionamento
MVC	– Model-View-Controller
PAN	– Personal Area Network
PDF	– Portable Document Format
PHP	– Hypertext Preprocessor
SPA	– Single Page Application
SQL	– Structured Query Language
TCC	– Trabalho de Conclusão de Curso
WAN	– Wide Area Network
WLAN	– Wireless Local Area Network
WMAN	– Wireless Metropolitan Area Network

SUMÁRIO

1 INTRODUÇÃO

Este trabalho de conclusão de curso, estruturado como um Portfólio de Projetos Acadêmicos, tem como objetivo explorar e apresentar a trajetória de aprendizado prático e teórico nas disciplinas essenciais do curso de Análise e Desenvolvimento de Sistemas (ADS), com destaque para os cursos de Linguagem de Programação (C++), Estrutura de Dados, Programação Orientada a Objetos Avançadas, Redes de Computadores e Tópicos Especiais em Informática.

No segundo semestre, a disciplina de Linguagem de Programação aprofundou o conhecimento dos fundamentos lógicos com a exploração da Linguagem C. Criada por Dennis M. Ritchie e Ken Thompson, esta linguagem foi escolhida pela sua capacidade de gerar código eficiente e robusto e por permitir o entendimento da programação de baixo nível, preparando o terreno para linguagens de maior nível. A aplicação prática desses conceitos deu-se através de um projeto Sistema de Recebimento, que simulou o faturamento e controle de aluguéis para uma locadora. O desafio consistiu em implementar a lógica de negócios, incluindo cálculos de multas por atraso e perda, e a estipulação de taxas de reposição e faturamento anual. O desenvolvimento foi estruturado em etapas (declaração, entrada de dados, cálculos e saída), o que facilitou a organização, a legibilidade e a manutenção do código, habilidades essenciais para um desenvolvedor. Este projeto consolidou a aplicação da sintaxe da linguagem C, o uso de tipos de dados e a implementação de estruturas de controle para solucionar um problema real do contexto empresarial.

No terceiro semestre, na disciplina de Estruturas de Dados, foi possível aprofundar o conhecimento dos alunos, focando na representação eficiente de informações e na criação de algoritmos eficazes. Foram estudados elementos essenciais como listas encadeadas, ponteiros, funções de *strings* e métodos de gerenciamento de arquivos. A culminação da aprendizagem foi o desenvolvimento de um sistema para gestão de livraria, que ilustrou a necessidade de aplicar diferentes estruturas de dados de forma estratégica para resolver demandas específicas. A aplicação prática desse conhecimento resultou no desenvolvimento de um sistema de gerenciamento de livraria, o que reforçou a importância de escolher e utilizar apropriadamente as estruturas de dados para solucionar problemas específicos. Este projeto consolidou a teoria e preparou os estudantes para desafios práticos em sistemas.

No quarto semestre, a disciplina de Programação Orientada a Objetos Avançada ofereceu a base teórica e prática para o desenvolvimento de soluções mais estruturadas e modulares. Embora a plataforma Arduino utilize uma linguagem baseada em C/C++, o estudo nesta disciplina permitiu a aplicação de conceitos de abstração, modularização e uso de código na construção do firmware. Este conhecimento foi aplicado no projeto Circuitos em Arduino, o qual focou na elaboração e montagem de configurações que integram hardware e software. O projeto prático incluiu a simulação de um circuito com sensor de presença e outros com sensores de proximidade (com e sem display). Para a execução, foi utilizado o ambiente virtual Tinkercad, que permitiu o prototipamento seguro com a placa Arduino Uno R3 e Protoboard. O estudo exigiu a aplicação de lógica complexa para controlar o ciclo de ativação de componentes (LEDs e Buzzer), além da conversão de pulsos ultrassônicos em distâncias (polegadas e centímetros), como no circuito do sensor de proximidade.

No quinto semestre, a disciplina de Redes de Computadores se mostrou fundamental para o desenvolvimento de competências essenciais na área de infraestrutura de TI. O conteúdo abordado incluiu a compreensão dos tipos de comunicação (Simplex, Half-Duplex e Full-Duplex), as classificações de redes e o funcionamento de dispositivos cruciais como *Switch*, *Router*, Web Server e os protocolos DHCP e DNS. Para a aplicação prática dessa teoria, utilizou-se o Cisco Packet Tracer, uma ferramenta de simulação que permitiu a criação de ambientes de rede simulados. Essa prática foi crucial para a experimentação de topologias e a configuração de dispositivos, solidificando o conhecimento sobre o fluxo de dados e a comunicação entre sistemas. A importância desta disciplina reside em fornecer o alicerce de conectividade para que os sistemas de softwares desenvolvidos, como o projeto web, possam operar de forma eficiente e segura, garantindo que o aluno compreenda o ambiente de implantação e as limitações de infraestrutura.

No sexto semestre, a disciplina de Tópicos Especiais em Informática atuou como o ponto de convergência de todos os conhecimentos adquiridos. Este módulo foi dedicado à integração de tecnologias e à solução de um desafio de desenvolvimento de software de ponta a ponta: a criação de uma Aplicação Web para Gerenciamento de Usuários. O projeto exigiu a unificação de habilidades em HTML, CSS e Bootstrap (para o front-end e estilização), PHP (para a lógica de back-end) e MySQL (para a persistência de dados). A aplicação desenvolvida implementa o ciclo completo de CRUD e, crucialmente, um controle de acesso por níveis de permissão

(Admin, Normal e Leitura). Este sistema prático demonstrou a proficiência na manipulação de bancos de dados, no tratamento de dados via scripts e na aplicação de lógica de segurança para restringir funcionalidades e proteger o sistema. A conclusão deste projeto em Tópicos Especiais atesta a capacidade do aluno de atuar em todas as fases do ciclo de vida do desenvolvimento de software, entregando soluções funcionais e seguras, prontas para um ambiente de produção.

A seleção desses pilares temáticos — abrangendo Programação Estruturada (C), Estruturas de Dados, Programação Orientada a Objetos, Redes de Computadores e Desenvolvimento Web Full-Stack — é integralmente justificada pela sua importância crítica na formação de um profissional de Análise e Desenvolvimento de Sistemas. Dominar a lógica de programação, modelar o raciocínio em estruturas de dados eficientes, garantir a persistência segura das informações, compreender a infraestrutura de comunicação em rede e integrar front-end e back-end são habilidades que, juntas, capacitam o estudante a enfrentar o ciclo completo de desenvolvimento de software.

A metodologia didática adotada, pautada na combinação de fundamentos teóricos robustos com a prática intensa em projetos (como o Sistema de Recebimento, a Gestão de Livraria, os Circuitos em Arduino e a Aplicação Web CRUD), permitiu a consolidação do aprendizado de forma progressiva e aplicada. Os resultados obtidos em cada etapa destacam a importância de uma base sólida para a criação de sistemas modulares, eficientes, seguros e preparados para operar em ambientes de rede modernos.

Para formalizar e tornar acessível este percurso de aprendizado e suas realizações, este trabalho foi estruturado como um Portfólio Digital de Projetos Acadêmicos. Desenvolvido com HTML, CSS e JavaScript, este portfólio organiza e expõe de maneira responsiva os principais projetos e as competências desenvolvidas ao longo do curso. Desta forma, o trabalho não apenas documenta, mas também demonstra a proficiência na entrega de soluções funcionais e a aptidão para atuar em todas as camadas da arquitetura de software.

2 SISTEMA DE RECEBIMENTO

Este capítulo aborda o desenvolvimento de uma aplicação em linguagem de programação C, realizada na disciplina de Linguagem de Programação, a fim de acrescentar conhecimento e conceitos essenciais para a aprendizagem. Conforme apresentado por Forbellone e Eberspacher (2005) em *Lógica de Programação*, “o estudo da lógica de programação fornece suporte teórico importante para a compreensão das estruturas de controle, algoritmos e resolução de problemas”, auxiliando na construção de códigos mais eficientes e claros.

2.1 LINGUAGEM C

A linguagem C é um marco da programação, criada em 1972 por Dennis M. Ritchie e Ken Thompson nos Laboratórios Bell. Conforme Ritchie (1993), “a criação da linguagem buscava gerar código eficiente e robusto, com uma sintaxe simples e poderosa”. A linguagem C é ideal para quem deseja compreender a programação de baixo nível, permitindo o controle direto da memória por meio do uso de ponteiros, o que resulta em programas mais rápidos e eficientes.

Embora não seja a linguagem mais utilizada atualmente, sua influência é imensa. Muitas linguagens populares, como Java, C++, Python e JavaScript foram diretamente influenciadas por C, herdando conceitos como tipagem de dados, estruturas de controle e manipulação de memória. Como afirma Ritchie (1993), “o legado da linguagem continua a moldar a programação moderna”.

2.2 DESENVOLVIMENTO

Este projeto, desenvolvido para a disciplina de Linguagem de Programação, foi idealizado com um objetivo claro: colocar em prática conhecimentos em linguagem C para a criação de uma aplicação funcional. O propósito estabelecido foi a simulação de um sistema de faturamento e controle de aluguéis para uma locadora. Ao longo do processo, foi possível lidar com diversos cenários, como o cálculo de multas por atraso e a reposição de itens perdidos.

O sistema opera com algumas regras predefinidas:

a) Multa por atraso: 10% da quantidade de fitas alugadas.

b) Multa por perda: 2% da quantidade total de fitas.

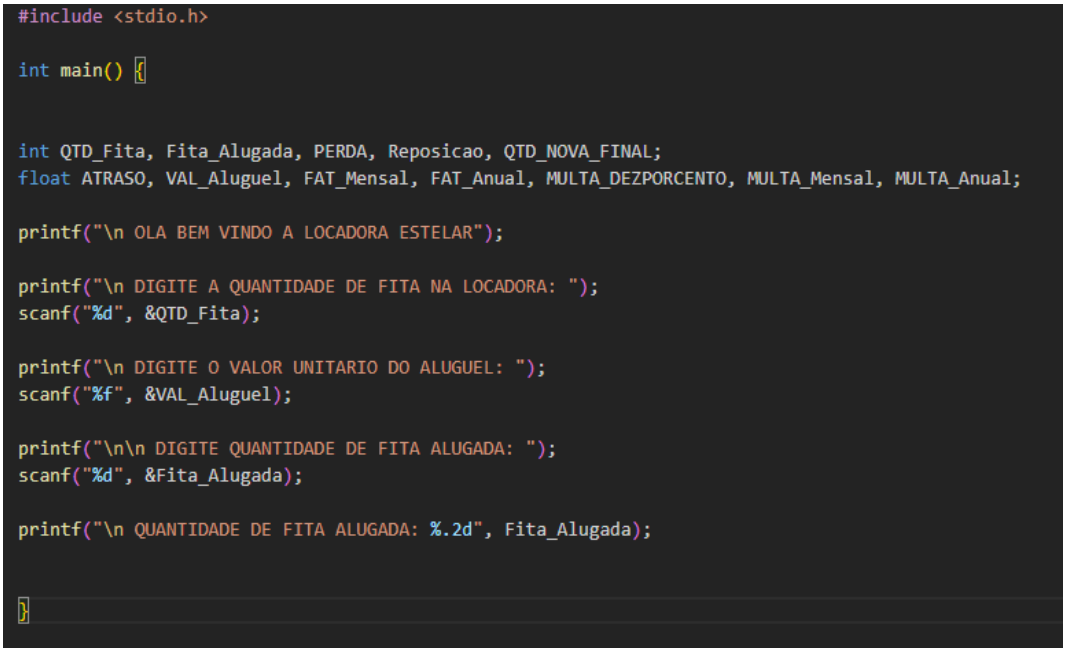
c) Valor da multa: 10% do valor do aluguel.

d) Taxa de reposição: 10% da quantidade de fitas.

O desenvolvimento do código foi organizado em quatro etapas principais.

A primeira etapa, apresentada neste trabalho, concentrou-se na declaração das variáveis e na coleta das informações necessárias para o processamento dos cálculos.

Figura 2.1 Estrutura de declaração e entrada de dado



```c
#include <stdio.h>

int main() {


int QTD_Fita, Fita_Alugada, PERDA, Reposicao, QTD_NOVA_FINAL;
float ATRASO, VAL_Aluguel, FAT_Mensal, FAT_Anual, MULTA_DEZPORCENTO, MULTA_Mensal, MULTA_Anual;

printf("\n OLA BEM VINDO A LOCADORA ESTELAR");

printf("\n DIGITE A QUANTIDADE DE FITA NA LOCADORA: ");
scanf("%d", &QTD_Fita);

printf("\n DIGITE O VALOR UNITARIO DO ALUGUEL: ");
scanf("%f", &VAL_Aluguel);

printf("\n\n DIGITE QUANTIDADE DE FITA ALUGADA: ");
scanf("%d", &Fita_Alugada);

printf("\n QUANTIDADE DE FITA ALUGADA: %.2d", Fita_Alugada);


}
```

Fonte: Elaborado pelo Próprio Autor

Na segunda etapa, são realizados os cálculos do faturamento mensal e a estipulação do faturamento anual com base no faturamento mensal.

Figura 2.2 Estrutura de Cálculos

```c
// Declaração de variáveis para faturamento mensal e anual
float fitas_alugadas = 100.0;  // Exemplo: 100 fitas alugadas
float valor_aluguel = 5.0;     // Exemplo: R$ 5,00 por aluguel

// Cálculo do faturamento mensal
float faturamento_mensal = fitas_alugadas * valor_aluguel;

// Cálculo do faturamento anual
float faturamento_anual = faturamento_mensal * 12;

// Impressão dos resultados
printf("O faturamento mensal é: R$ %.2f\n", faturamento_mensal);
printf("O faturamento anual é: R$ %.2f\n", faturamento_anual);

return 0;
```

Fonte: Elaborado pelo Próprio Autor

Na terceira etapa, é efetuado o cálculo do valor total das multas (mensais e anuais) referentes às fitas de vídeo alugadas com atraso.

Figura 2.3 Estrutura de cálculos de multas/atraso

```c
// Definindo valores de exemplo
float fitas_alugadas = 100.0;
float valor_aluguel = 5.0;

// Cálculo da quantidade de fitas com atraso
float fitas_atrasadas = fitas_alugadas / 10.0;

// Cálculo do valor da multa por fita (10% do valor do aluguel)
float valor_multa_por_fita = valor_aluguel * 0.10; // 10% é 0.10

// Cálculo do valor total das multas mensais
float multas_mensais_totais = fitas_atrasadas * valor_multa_por_fita;

// Cálculo do valor total das multas anuais
float multas_anuais_totais = multas_mensais_totais * 12;

// Impressão dos resultados
printf("Quantidade de fitas com atraso: %.0f\n", fitas_atrasadas);
printf("Valor da multa por fita: R$ %.2f\n", valor_multa_por_fita);
printf("Valor total das multas mensais: R$ %.2f\n", multas_mensais_totais);
printf("Valor total das multas anuais: R$ %.2f\n", multas_anuais_totais);
```

Fonte: Elaborado pelo Próprio Autor

Na quarta e última etapa, são realizados os cálculos de perdas e de reposição das fitas.

Figura 2.4 Cálculos de perdas e reposição das fitas

```c
// Definindo a quantidade inicial de fitas
float quantidade_inicial_fitas = 500.0; // Exemplo: 500 fitas

// Cálculo da perda de fitas (2% da quantidade inicial)
// O valor '2' é na verdade 2%
float fitas_perdidas = quantidade_inicial_fitas * 0.02; // 2% é 0.02

// Cálculo das fitas compradas para reposição
float fitas_repostas = quantidade_inicial_fitas / 10.0; // 10% da quantidade inicial

// Cálculo da quantidade final de fitas após perdas e reposição
float quantidade_final = (quantidade_inicial_fitas - fitas_perdidas) + fitas_repostas;

// Impressão dos resultados
printf("Quantidade de fitas perdidas: %.0f\n", fitas_perdidas);
printf("Quantidade de fitas repostas: %.0f\n", fitas_repostas);
printf("Quantidade total de fitas no final do ano: %.0f\n", quantidade_final);

return 0;
```

Fonte: Elaborado pelo Próprio Autor

2.3 CONCLUSÃO

O tema da disciplina mostrou-se fundamental para o aprendizado, especialmente por direcionar a elaboração do código em etapas. Essa abordagem proporcionou benefícios importantes ao desenvolvimento do projeto:

a) Melhor organização: A divisão do trabalho em partes facilitou o gerenciamento de cada fase.

b) Maior compreensão: A estrutura sequencial permitiu um entendimento mais claro de cada funcionalidade.

c) Código mais limpo: A organização contribuiu para a criação de um código mais legível e de fácil manutenção.

O resultado final consiste em uma aplicação capaz de calcular, com base nas informações fornecidas, a quantidade de fitas alugadas e perdidas no ano. Além disso, a aplicação determina o valor total faturado e apresenta a quantidade de fitas que necessitam de reposição.

3 SISTEMA DE GERENCIAMENTO DE LIVRARIA

3.1 ESTRUTURA DE DADOS

As estruturas de dados são fundamentais para organizar e manipular informações de forma eficiente, permitindo a criação de algoritmos mais eficazes. Entre elas, encontram-se *arrays*, listas, pilhas, filas, árvores e grafos. A escolha da estrutura de dados adequada é essencial para otimizar o desempenho de um programa. Por exemplo, uma lista ligada apresenta eficiência na inserção e remoção de elementos, enquanto uma árvore binária de busca oferece melhor desempenho na localização de dados em listas ordenadas.

A disciplina de Estrutura de Dados, ministrada no terceiro semestre do curso de ADS, apresenta a aplicação prática desses conceitos. Um projeto de livraria demonstrou a importância do uso de listas para organizar livros, ponteiros para gerenciar a memória e funções de *string* e arquivo para lidar com informações e garantir a persistência dos dados.

O estudo das estruturas de dados mantém relação direta com os algoritmos e é indispensável para o desenvolvimento de soluções de software otimizadas. Além disso, conforme Elmasri e Navathe (2011), “o entendimento de como dados são estruturados e manipulados é essencial para o projeto e gerenciamento de bancos de dados”, dando ênfase de que e necessário escolher adequadamente a estrutura de dados para garantir eficiência, integridade e consistência nas aplicações.

3.2 DESENVOLVIMENTO

O objetivo consistiu no desenvolvimento de um sistema de gerenciamento de livraria projetado para automatizar as operações de uma loja de livros. O sistema possibilita a inclusão, listagem, consulta, atualização e remoção de registros relacionados a livros, clientes, autores e vendas. O projeto foi implementado utilizando as técnicas e conhecimentos adquiridos em linguagem C.

O código utiliza a diretiva (`#define`) para criar constantes simbólicas (LIVROS, CLIENTES, VENDAS). Essa abordagem evita a necessidade de escrever os nomes dos arquivos por completo sempre que são chamados, o que aumenta a manutenibilidade e a clareza do código.

Figura 3.1 Definição de Bibliotecas e variáveis

```c
// Inclui as bibliotecas padrão do C
#include <stdio.h>    // Para funções de entrada e saída (ex: printf, scanf)
#include <string.h>   // Para manipulação de strings (ex: strcpy, strlen)
#include <ctype.h>    // Para funções de verificação de caracteres (ex: isalpha, isdigit)

// Define constantes para os nomes dos arquivos.
// Isso torna o código mais fácil de manter,
// pois se o nome do arquivo mudar, você só
// precisa alterá-lo neste local.
#define LIVROS "livros.dat"
#define CLIENTES "clientes.dat"
#define VENDAS "vendas.dat"

// O restante do código, como a declaração de funções e a função main(), viria aqui.
```

Fonte: Elaborado pelo Próprio Autor

Para organizar as informações do projeto, são utilizadas estruturas (`structs`). Em linguagem C, uma *struct* permite a criação de um tipo de dado capaz de agrupar variáveis com propósitos relacionados. Essa prática é essencial para garantir clareza e eficiência na manipulação de dados complexos. Como exemplo, a *struct* (`regLivro`) funciona como um molde que contém todos os atributos necessários para representar um livro-código, nome, preço e estoque. Ao utilizar tal estrutura, todas essas informações passam a ser tratadas como uma única entidade. Com as *structs* definidas, as constantes declaradas e as bibliotecas incluídas, inicia-se a implementação das funções que compõem o funcionamento do programa.

Figura 3.2 Definição do Formato das Struct´s

```
// Define uma estrutura para representar um livro.
// Esta "ficha" armazena todas as informações importantes sobre um livro.
struct regLivro {
    int codigo;        // Código único de identificação do livro.
    char nome[30];     // Título do livro, com até 30 caracteres.
    float preco;       // Preço do livro em formato decimal.
    int estoque;       // Quantidade de unidades disponíveis em estoque.
};

// ---

// Define uma estrutura para representar um cliente.
// Esta "ficha" agrupa os dados pessoais e de contato de um cliente.
struct regCliente {
    int codigo;         // Código único de identificação do cliente.
    char nome[50];      // Nome completo do cliente, com até 50 caracteres.
    char endereco[50];  // Endereço residencial do cliente.
    char fone[15];      // Número de telefone para contato.
    char email[50];     // Endereço de e-mail do cliente.
};

// ---

// Define uma estrutura para representar uma venda ou um pedido.
// Esta "ficha" associa um cliente a um livro e a uma quantidade vendida.
struct regVenda {
    int codPedido;    // Código único do pedido ou da venda.
    int codCliente;   // Código do cliente que realizou a compra.
    int codLivro;     // Código do livro que foi vendido.
    int quantidade;   // Quantidade de unidades do livro vendidas neste pedido.
};
```

Fonte: Elaborado pelo Próprio Autor

A função `cadastrarLivro` Figura 3.3 tem como objetivo realizar a coleta de dados da interface e o cadastro de um livro. No início da função é realizada a abertura do arquivo onde a informação será salva e criada uma *struct* de `regLivro`. Posteriormente, são coletadas as informações para registro e, por fim, é realizada a gravação das informações dentro do arquivo, seguindo-se o fechamento.

Figura 3.3 Função cadastrar livro

```c
void cadastrarLivro() {
    // Declara um ponteiro de arquivo para manipular o arquivo de livros.
    FILE *fplivro;
    // Declara uma variável do tipo struct regLivro para armazenar os dados temporariamente.
    struct regLivro livro;
    // Declara um array de caracteres para armazenar a opção do usuário (S/N).
    char opca[1];
    // Solicita e lê o código do livro.
    printf("\n Digite o Codigo: ");
    fflush(stdin); // Limpa o buffer de entrada.
    scanf("%i", &livro.codigo);

    // Solicita e lê o nome do livro.
    printf("\n Digite o Nome: ");
    fflush(stdin);
    gets(livro.nome); // Lê a linha completa, incluindo espaços.

    // Solicita e lê o preço do livro.
    printf("\n Digite o Preco: ");
    fflush(stdin);
    scanf("%f", &livro.preco);

    // Solicita e lê a quantidade em estoque.
    printf("\n Digite a quantidade: ");
    fflush(stdin);
    scanf("%i", &livro.estoque);

    // Pergunta ao usuário se ele deseja salvar os dados.
    printf("Deseja Salvar (S/N) ?");
    fflush(stdin);
    scanf("%c", &opca);

    // Verifica se a opção foi 'N' (não) ou 'n'.
    if((opca[0] == 'N') || (opca[0] == 'n')) {
        printf("\n Cadastro Cancelado");
        return; // Sai da função sem salvar.
    }
// Abre o arquivo de livros em modo de adição binária ("ab").
// O modo "a" garante que o novo registro será adicionado no final do arquivo.
// O modo "b" é para lidar com arquivos binários.
fplivro = fopen(LIVROS, "ab");

// Escreve os dados da struct 'livro' no arquivo.
// fwrite(endereço_da_variavel, tamanho_da_variavel, quantidade, ponteiro_do_arquivo);
fwrite(&livro, sizeof(livro), 1, fplivro);

// Exibe uma mensagem de sucesso para o usuário.
printf("\b Livro registrado com sucesso");

// Fecha o arquivo. Isso é crucial para garantir que os dados sejam salvos
// e para liberar os recursos do sistema.
fclose(fplivro);
```

Fonte: Elaborado pelo Próprio Autor

A função `cadastrarCliente` figura 3.4 atua como o módulo responsável pela entrada de dados e pela persistência dos registros de novos clientes. A lógica interna abrange desde a abertura do arquivo para gravação até a criação da *struct* `regCliente`.Em seguida, as informações são coletadas e gravadas no arquivo, encerrando-se o fluxo com o fechamento adequado do recurso

Figura 3.4 Função Cadastrar Cliente

```c
void cadastrarCliente() {
    // Declara um ponteiro de arquivo para manipular o arquivo de clientes.
    FILE *fpclientes;
    // Declara uma variável do tipo struct regCliente para armazenar os dados.
    struct regCliente cliente;
    // Declara um caractere para armazenar a opção do usuário (S/N).
    char opcao;

    // Solicita e lê o código de identificação do cliente.
    printf("\n Digite o Codigo: ");
    fflush(stdin); // Limpa o buffer de entrada.
    scanf("%i", &cliente.codigo);

    // Solicita e lê o nome do cliente.
    printf("\n Digite o Nome: ");
    fflush(stdin);
    gets(cliente.nome);
    // Nota: O uso de gets() é obsoleto e inseguro.

    // Solicita e lê o endereço.
    printf("\n Digite o Endereco: ");
    fflush(stdin);
    gets(cliente.endereco);

    // Solicita e lê o telefone.
    printf("\n Digite o Telefone: ");
    fflush(stdin);
    gets(cliente.fone);

    // Solicita e lê o e-mail.
    printf("\n Digite o Email: ");
    fflush(stdin);
    gets(cliente.email);

    // Pergunta ao usuário se deseja salvar o novo registro.
    printf("Deseja Salvar (S/N) ?");
    fflush(stdin);
    scanf("%c", &opcao);
    // Verifica se o usuário optou por NÃO salvar (N ou n).
    if((opcao == 'N') || (opcao == 'n')) {
        printf("\t_ Cadastro Cancelado");
        return; // Encerra a função, cancelando o cadastro.
    }

    // --- GRAVAÇÃO NO ARQUIVO ---

    // Abre o arquivo CLIENTES em modo de adição binária ("ab").
    // O modo "ab" garante que o novo registro será adicionado no final.
    fpclientes = fopen(CLIENTES, "ab");

    // Escreve os dados da struct 'cliente' no arquivo.
    // fwrite(endereço, tamanho_da_struct, quantidade_de_itens, ponteiro_do_arquivo)
    fwrite(&cliente, sizeof(cliente), 1, fpclientes);

    // Exibe mensagem de sucesso.
    printf("\n Cadastro Realizado com Sucesso !!!");

    // Fecha o arquivo para salvar os dados e liberar o recurso.
    fclose(fpclientes);
}
```

Fonte: Elaborado pelo Próprio Autor

A função `LocalizarCliente` Figura 3.5 é um método essencial de recuperação de dados, projetado para trabalhar diretamente com a estrutura `regCliente`. Sua arquitetura é orientada a *struct*, garantindo que o dado retornado seja um objeto completo e tipado. Mecanismo de Recuperação A operação se concentra em uma busca linear dentro do arquivo binário de clientes. O processo é otimizado: após declarar as variáveis locais e abrir o arquivo para leitura, a função executa uma varredura condicional. A cada iteração, ela tenta ler um bloco de dados do tamanho da *struct* e verifica se o código corresponde. Esta abordagem garante a eficiência do sistema: assim que o registro é encontrado, o *loop* é finalizado pelo *flag* de controle, evitando leituras desnecessárias até o final do arquivo.

Figura 3.5 Função Localizar Cliente

```c
// Define a função LocalizarCliente. Ela recebe o código que deve ser procurado
// e retorna uma struct regCliente.
struct regCliente LocalizarCliente(int codCliente) {
    // Declara um ponteiro de arquivo para manipular o arquivo de clientes.
    FILE *fpcliente;
    // Declara uma struct para armazenar os dados do cliente lido a cada vez.
    struct regCliente cliente;
    // 'posicao' funciona como um flag (bandeira). 0 = não encontrado; 1 = encontrado.
    int posicao = 0;

    // Abre o arquivo CLIENTES no modo de Leitura Binária ("rb").
    fpcliente = fopen(CLIENTES, "rb");

    // Loop de busca: Continua enquanto o cliente não for encontrado (posicao == 0)
    // E enquanto houver registros para ler no arquivo (fread() retorna 1).
    while((posicao == 0) && (fread(&cliente, sizeof(cliente), 1, fpcliente) == 1)) {
        // Verifica se o código do cliente lido é igual ao código procurado.
        if(cliente.codigo == codCliente) {
            // Se encontrar, muda a flag para 1, encerrando o loop na próxima iteração.
            posicao = 1;
        }
    }
    // Fecha o arquivo após a busca (encontrando ou não).
    fclose(fpcliente);

    // Se a flag 'posicao' ainda for 0, significa que o cliente não foi encontrado.
    if (posicao == 0) {
        // Define o código como -1 para sinalizar o erro ou a falha na busca.
        cliente.codigo = -1;
    }

    // Retorna a struct cliente. Ela conterá os dados encontrados ou o código -1.
    return cliente;
}
```

Fonte: Elaborado pelo Próprio Autor

A função `localizarVenda` Figura 3.6 tem como sua característica principal ser uma função específica para *struct*. No qual, por meio do código da venda, tem por objetivo localizar e retornar as informações de uma venda registrada nos arquivos. Na função da figura 3.6 é realizada uma busca dentro do arquivo de cliente. A função inicia com declaração de informações que são usadas durante sua a execução, antes do processamento de informações ser realizado, uma validação é feita no arquivo venda para que o mesmo seja aberto e consultado. Posteriormente, executa-se um laço de repetição percorrendo todo o arquivo. Uma vez que a informação é localizada no arquivo de cliente, a leitura é interrompida e as informações encontradas são colocadas em uma *struct* para a sua exibição caso a informação não seja encontrada a aplicação retornará um número negativo e o sistema assim sabendo que não existe registro.

Figura 3.6 Função Localizar Venda

```
// Define a função LocalizarVenda. Ela recebe o código da venda (codVenda)
// e retorna uma struct regVenda (o registro da venda).
struct regVenda LocalizarVenda(int codVenda) {
    // Declara um ponteiro de arquivo para manipulação do arquivo de vendas.
    FILE *fpvenda;
    // Declara uma struct para armazenar os dados do registro lido do arquivo.
    struct regVenda venda;
    // 'posicao' é um flag: 0 (não encontrado) ou 1 (encontrado).
    int posicao = 0;

    // Abre o arquivo VENDAS ("vendas.dat") em modo de Leitura Binária ("rb").
    if((fpvenda = fopen(VENDAS, "rb")) == NULL) {
        // Verifica se houve ERRO ao abrir o arquivo. Se sim, imprime uma mensagem.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", VENDAS);
    }
    // Inicializa a flag novamente antes do loop (apesar de já ser 0, reforça a intenção).
    posicao = 0;
    // Loop de busca sequencial: Continua enquanto a venda não for encontrada (posicao == 0)
    // E a leitura de um novo registro for bem-sucedida (fread() == 1).
    // Nota: O loop é interrompido imediatamente após o primeiro registro correspondente.
    while((posicao == 0) && (fread(&venda, sizeof(venda), 1, fpvenda) == 1)) {
        // Compara o código do pedido lido com o código procurado.
        if(venda.codPedido == codVenda) {
            // Se houver correspondência, muda a flag para 1, sinalizando sucesso na busca.
            posicao = 1;
        }
    }
    // Fecha o arquivo. Essencial para liberar o recurso.
    fclose(fpvenda);
    // Bloco de tratamento de erro e sinalização:
    // Se a flag for 0, significa que o registro não foi localizado após a varredura.
    if(posicao == 0) {
        // Define o campo chave com -1 para sinalizar a falha da busca.
        venda.codPedido = -1;
    }
    // Retorna a struct de venda (encontrada ou com o código de falha -1).
    return venda;
}
```

Fonte: Elaborado pelo Próprio Autor

A função `listarLivro` Figura 3.7 tem como sua característica principal de realizar a abertura do arquivo livro, coletar todos os seus registros exibi-los na tela.

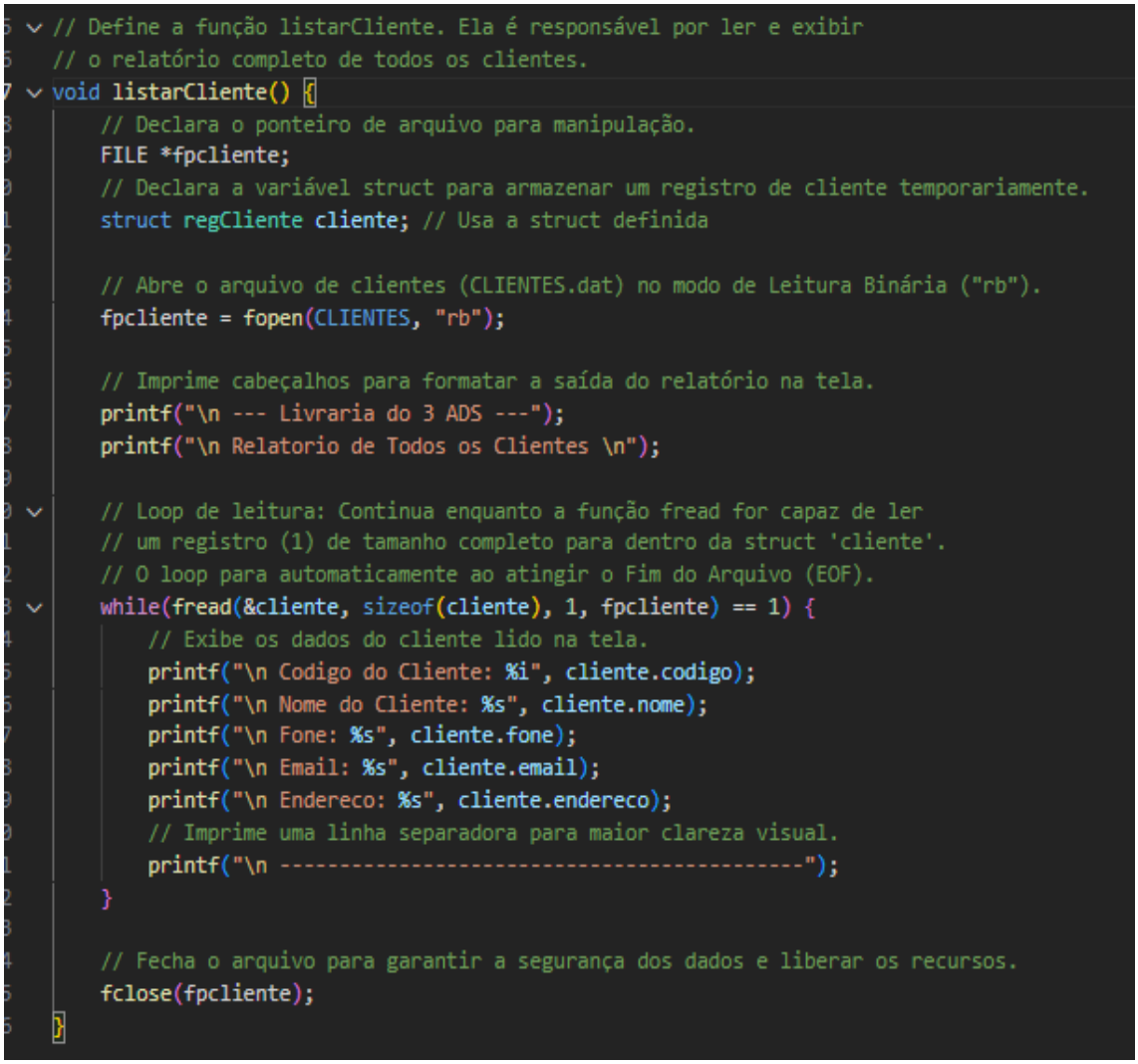
Figura 3.7 Função Listar Livros

```c
// Define a função listarLivro. Sua finalidade é ler e exibir todos os livros.
void listarLivro() {
    // Declara o ponteiro de arquivo.
    FILE *fplivro;
    // Declara a variável struct para armazenar um registro de livro temporariamente.
    struct regLivro livro;

    // Abre o arquivo de livros (LIVROS.dat) no modo de leitura binária ("rb").
    fplivro = fopen(LIVROS, "rb");

    // Imprime o cabeçalho do relatório para formatação.
    printf("\n --- Livraria do 3 ADS ---");
    printf("\n Relatorio de Todos Livros \n");

    // Inicia o loop para ler o arquivo:
    // Continua enquanto a função fread conseguir ler um registro (1) de tamanho completo.
    // O loop para automaticamente ao atingir o Fim do Arquivo (EOF).
    while(fread(&livro, sizeof(livro), 1, fplivro) == 1) {
        // Exibe os dados do livro lido na tela.
        printf("\n Codigo -> %i", livro.codigo);
        printf("\n Titulo -> %s", livro.nome);
        // Exibe o preço formatado com duas casas decimais.
        printf("\n Preco -> %.2f", livro.preco);
        printf("\n Estoque -> %i", livro.estoque);
        // Imprime uma linha separadora para melhor visualização.
        printf("\n --------------------------------");
    }

    // Fecha o arquivo para garantir a segurança e liberar o recurso.
    fclose(fplivro);
}
```

Fonte: Elaborado pelo Próprio Autor

A função `listarCliente` Figura 3.8 é muito parecida à função `listarLivros` Figura 3.7, e tem como objetivo de realizar abertura do arquivo Cliente e trazer todos os seus registros.

Figura 3.8 Listar Cliente

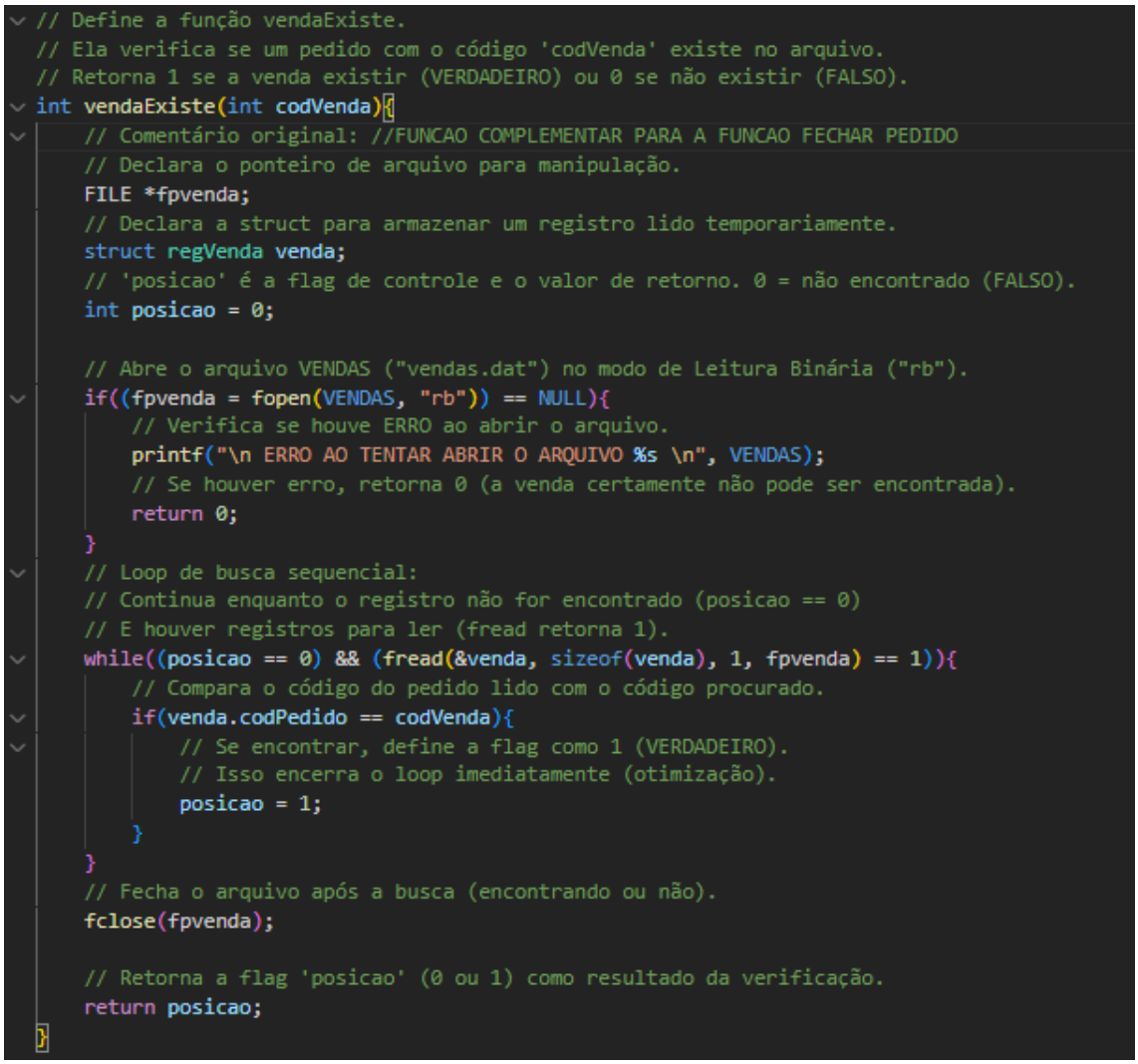


```c
// Define a função listarCliente. Ela é responsável por ler e exibir
// o relatório completo de todos os clientes.
void listarCliente() {
    // Declara o ponteiro de arquivo para manipulação.
    FILE *fpcliente;
    // Declara a variável struct para armazenar um registro de cliente temporariamente.
    struct regCliente cliente; // Usa a struct definida

    // Abre o arquivo de clientes (CLIENTES.dat) no modo de Leitura Binária ("rb").
    fpcliente = fopen(CLIENTES, "rb");

    // Imprime cabeçalhos para formatar a saída do relatório na tela.
    printf("\n --- Livraria do 3 ADS ---");
    printf("\n Relatorio de Todos os Clientes \n");

    // Loop de leitura: Continua enquanto a função fread for capaz de ler
    // um registro (1) de tamanho completo para dentro da struct 'cliente'.
    // O loop para automaticamente ao atingir o Fim do Arquivo (EOF).
    while(fread(&cliente, sizeof(cliente), 1, fpcliente) == 1) {
        // Exibe os dados do cliente lido na tela.
        printf("\n Codigo do Cliente: %i", cliente.codigo);
        printf("\n Nome do Cliente: %s", cliente.nome);
        printf("\n Fone: %s", cliente.fone);
        printf("\n Email: %s", cliente.email);
        printf("\n Endereco: %s", cliente.endereco);
        // Imprime uma linha separadora para maior clareza visual.
        printf("\n ----------------------------------------");
    }

    // Fecha o arquivo para garantir a segurança dos dados e liberar os recursos.
    fclose(fpcliente);
}
```

Fonte: Elaborado pelo Próprio Autor

A Função `vendaExiste` Figura 3.9 é um utilitário de validação crucial no sistema. Ela se destaca por ser uma função puramente de consulta, que retorna um valor inteiro para indicar a existência ou não de um registro, seguindo a lógica booleana: 1 para êxito (Verdadeiro) e 0 para falha (Falso).

Figura 3.9 Função VendaExiste

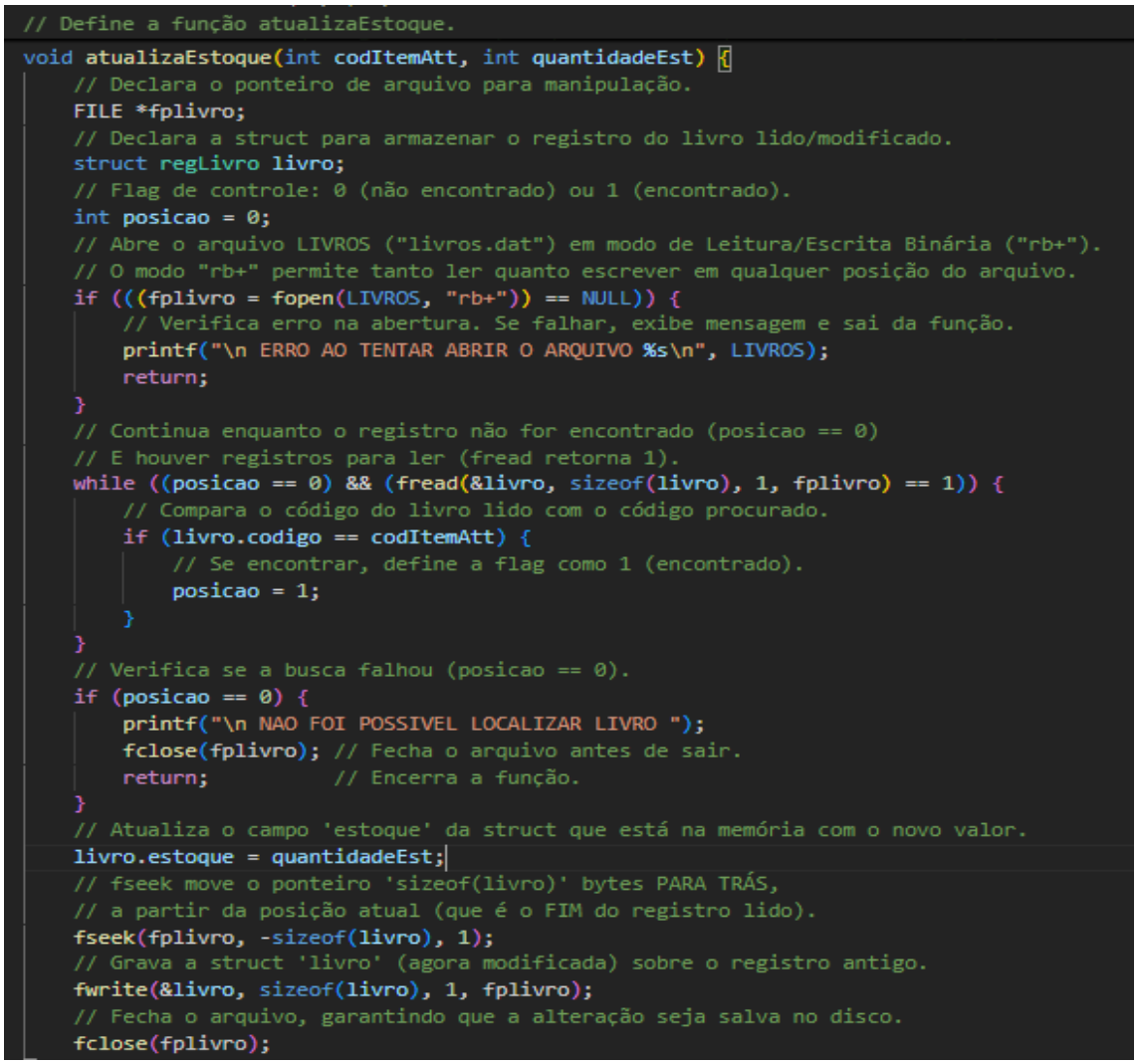


```c
// Define a função vendaExiste.
// Ela verifica se um pedido com o código 'codVenda' existe no arquivo.
// Retorna 1 se a venda existir (VERDADEIRO) ou 0 se não existir (FALSO).
int vendaExiste(int codVenda){
    // Comentário original: //FUNCAO COMPLEMENTAR PARA A FUNCAO FECHAR PEDIDO
    // Declara o ponteiro de arquivo para manipulação.
    FILE *fpvenda;
    // Declara a struct para armazenar um registro lido temporariamente.
    struct regVenda venda;
    // 'posicao' é a flag de controle e o valor de retorno. 0 = não encontrado (FALSO).
    int posicao = 0;

    // Abre o arquivo VENDAS ("vendas.dat") no modo de Leitura Binária ("rb").
    if((fpvenda = fopen(VENDAS, "rb")) == NULL){
        // Verifica se houve ERRO ao abrir o arquivo.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", VENDAS);
        // Se houver erro, retorna 0 (a venda certamente não pode ser encontrada).
        return 0;
    }
    // Loop de busca sequencial:
    // Continua enquanto o registro não for encontrado (posicao == 0)
    // E houver registros para ler (fread retorna 1).
    while((posicao == 0) && (fread(&venda, sizeof(venda), 1, fpvenda) == 1)){
        // Compara o código do pedido lido com o código procurado.
        if(venda.codPedido == codVenda){
            // Se encontrar, define a flag como 1 (VERDADEIRO).
            // Isso encerra o loop imediatamente (otimização).
            posicao = 1;
        }
    }
    // Fecha o arquivo após a busca (encontrando ou não).
    fclose(fpvenda);

    // Retorna a flag 'posicao' (0 ou 1) como resultado da verificação.
    return posicao;
}
```

Fonte: Elaborado pelo Próprio Autor

A função `atualizaEstoque` (apresentada na Figura 3.10) representa uma rotina de Manutenção de Dados (UPDATE). O objetivo da função é garantir a integridade do inventário, localizando um item específico (`codItemAtt`) e alterando seu campo de estoque (`quantidadeEst`) de forma permanente no arquivo binário.

Figura 3.10 Função AtualizarEstoque



```c
// Define a função atualizaEstoque.
void atualizaEstoque(int codItemAtt, int quantidadeEst) {
    // Declara o ponteiro de arquivo para manipulação.
    FILE *fplivro;
    // Declara a struct para armazenar o registro do livro lido/modificado.
    struct regLivro livro;
    // Flag de controle: 0 (não encontrado) ou 1 (encontrado).
    int posicao = 0;
    // Abre o arquivo LIVROS ("livros.dat") em modo de Leitura/Escrita Binária ("rb+").
    // O modo "rb+" permite tanto ler quanto escrever em qualquer posição do arquivo.
    if (((fplivro = fopen(LIVROS, "rb+")) == NULL)) {
        // Verifica erro na abertura. Se falhar, exibe mensagem e sai da função.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s\n", LIVROS);
        return;
    }
    // Continua enquanto o registro não for encontrado (posicao == 0)
    // E houver registros para ler (fread retorna 1).
    while ((posicao == 0) && (fread(&livro, sizeof(livro), 1, fplivro) == 1)) {
        // Compara o código do livro lido com o código procurado.
        if (livro.codigo == codItemAtt) {
            // Se encontrar, define a flag como 1 (encontrado).
            posicao = 1;
        }
    }
    // Verifica se a busca falhou (posicao == 0).
    if (posicao == 0) {
        printf("\n NAO FOI POSSIVEL LOCALIZAR LIVRO ");
        fclose(fplivro); // Fecha o arquivo antes de sair.
        return;          // Encerra a função.
    }
    // Atualiza o campo 'estoque' da struct que está na memória com o novo valor.
    livro.estoque = quantidadeEst;
    // fseek move o ponteiro 'sizeof(livro)' bytes PARA TRÁS,
    // a partir da posição atual (que é o FIM do registro lido).
    fseek(fplivro, -sizeof(livro), 1);
    // Grava a struct 'livro' (agora modificada) sobre o registro antigo.
    fwrite(&livro, sizeof(livro), 1, fplivro);
    // Fecha o arquivo, garantindo que a alteração seja salva no disco.
    fclose(fplivro);
```

Fonte: Elaborado pelo Próprio Autor

A Função Registrar Venda Figura 3.11 e Figura 3.12 é uma das funções mais complexas do seu sistema, pois ela não apenas coleta dados e salva um novo registro, mas também realiza validações cruzadas com outros arquivos (clientes) e utiliza funções de busca previamente definidas como (`LocalizarCliente`).

Figura 3.11 Função Registrar Venda 1

```c
void registrarVenda() {
    // Declaração de ponteiros para acesso aos 3 arquivos necessários.
    FILE *fplivro, *fpcliente, *fpvenda;
    // Declaração das structs que serão usadas para manipular os dados.
    struct regLivro livro;
    struct regCliente cliente;
    struct regVenda venda;
    // Variável para armazenar a opção do usuário (S/N).
    char opcao;
    int posicao, codVenda, codCliente, codLivro;
    // --- 1. COLETA DE DADOS INICIAIS ---
    // Solicita o código da nova venda.
    printf("\n Codigo Venda: ");
    fflush(stdin); scanf("%i", &codVenda);
    // Solicita o código do cliente para a validação.
    printf("\n Codigo do Cliente: ");
    fflush(stdin); scanf("%i", &codCliente);
    // --- 2. VALIDAÇÃO DO CLIENTE ---
    // Tenta localizar o cliente usando o código fornecido.
    cliente = LocalizarCliente(codCliente);
    // Verifica se a função LocalizarCliente retornou o código de erro (-1).
    if(cliente.codigo == -1) {
        printf("\n Nao foi possivel localizar Cliente");
        return; // Cancela a operação se o cliente não for encontrado.
    } else {
        // Se o cliente foi encontrado, exibe seu nome como feedback.
        printf("\n Cliente Localizado: %s", cliente.nome);
    }
    // --- 3. CONFIRMAÇÃO DA OPERAÇÃO ---
    printf("\n Confirma Cliente? (S/N): ");
    fflush(stdin); scanf("%c", &opcao);
    // Verifica se a operação foi cancelada.
    if(((opcao == 'N') || (opcao == 'n'))) {
        printf("\n Operacao Cancelada com Sucesso");
        return; // Sai da função.
    }
    // A Parte 2 da função (busca do livro e gravação) viria após este ponto.
}
```

Fonte: Elaborado pelo Próprio Autor

A Função Registrar Venda 2 finaliza o fluxo de vendas, incluindo a validação do livro, a confirmação do item, a gravação do novo registro de venda no arquivo e o loop de repetição para adicionar múltiplos itens ao mesmo pedido.

Figura 3.12 Função Registrar Venda 2

```
do {
    // Solicita o código do livro que será incluído no pedido.
    printf("\n Codigo do Livro: ");
    fflush(stdin); scanf("%i", &codLivro);
    // Chama a função de busca (assumida) para localizar o livro.
    livro = localizarLivro(CodLivro);

    // Verifica se o livro foi encontrado (código != -1).
    if (livro.codigo != -1) {
        // Exibe o nome e o preço do livro como feedback.
        printf("\n Nome do Livro: %s -- Preco: %.2f", livro.nome, livro.preco);
        // Primeira Confirmação: O usuário confirma que encontrou o livro certo.
        printf("\n Confirma Livro (S/N)? ");
        fflush(stdin); scanf("%c", &opcao);
        if((opcao == 'S') || (opcao == 's')) {
            // Segunda Confirmação: O usuário confirma que deseja registrar a venda deste item.
            printf("\n Registrar Venda deste Item? (S/N) ");
            fflush(stdin); scanf("%c", &opcao);
            if((opcao == 'S') || (opcao == 's')) {
                // --- 5. COLETA FINAL E PERSISTÊNCIA ---
                printf("\n Digite a Quantidade: ");
                fflush(stdin); scanf("%i", &venda.quantidade);
                // Atribui os códigos de controle (venda, livro, cliente) à struct de venda.
                venda.codpedido = codVenda;    // Código da Venda (Parte 1)
                venda.codLivro = livro.codigo; // Código do Livro (Parte 2)
                venda.codCliente = cliente.codigo; // Código do Cliente (Parte 1)
                // Abre o arquivo VENDAS em modo de Apêndice Binário ("ab").
                fpvenda = fopen(VENDAS, "ab");
                // Grava o novo registro de venda no final do arquivo.
                fwrite(&venda, sizeof(venda), 1, fpvenda);
                // Fecha o arquivo.
                fclose(fpvenda);
                printf("\n Item inserido com sucesso");
            } else {
                printf("Item cancelado");
            }
        } else {
            // Se a primeira confirmação for 'N'.
            printf("\n Livro nao Confirmado");
        }
    } else {
        // Se a busca retornar código -1.
        printf("\n Livro nao Confirmado");
    }
    // Pergunta se o usuário deseja adicionar mais um item ao pedido atual.
    printf("\n Deseja registrar outro item nesta venda? (S/N) ");
    fflush(stdin); scanf("%c", &opcao);
// O loop continua se a resposta for 'S' ou 's'.
} while ((opcao == 'S') || (opcao == 's'));
```

Fonte: Elaborado pelo Próprio Autor

Nesta primeira parte da função é realizada toda declaração de *struct* e informações usadas e validações de abertura dos arquivos. A função `ListarVenda` Figura 3.13 e Figura 3.14 é o módulo de Relatório Cruzado do sistema, responsável por gerar uma visualização detalhada de todas as vendas realizadas.

Figura 3.13 Função Listar Venda – Parte 1

```c
// Define a função ListarVenda. Esta função é um relatório complexo
// que lê os itens do arquivo VENDAS e consulta detalhes em CLIENTES e LIVROS.
void ListarVenda() {
    // Declaração dos ponteiros para os três arquivos necessários.
    FILE *fpvenda, *fplivro, *fpcliente, *fpctrlLivro;
    struct regVenda venda;
    struct regLivro ctrlLivro;
    struct regLivro livro;
    struct regCliente cliente;

    // Variáveis de controle para o loop e totalização.
    int codigoVenda, posicao = 0;
    int ctrlPedido = 0; // Usado para saber quando a venda muda (exibir cabeçalho).
    float valorTotal = 0;

    // --- 1. ABERTURA SEGURA DO ARQUIVO VENDAS ---
    // Tenta abrir o arquivo VENDAS ("vendas.dat") em modo de Leitura Binária ("rb").
    if(((fpvenda = fopen(VENDAS, "rb")) == NULL)){
        // Se houver erro, exibe a mensagem e sai da função.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", VENDAS);
        return;
    }
    // --- 2. ABERTURA SEGURA DO ARQUIVO CLIENTES ---
    // Tenta abrir o arquivo CLIENTES ("clientes.dat") em modo de Leitura Binária ("rb").
    if(((fpcliente = fopen(CLIENTES, "rb")) == NULL)){
        // Se houver erro, exibe a mensagem e sai da função.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", CLIENTES);
        return;
    }
    // --- 3. ABERTURA SEGURA DO ARQUIVO LIVROS ---
    // Tenta abrir o arquivo LIVROS ("livros.dat") em modo de Leitura Binária ("rb").
    if(((fplivro = fopen(LIVROS, "rb")) == NULL)){
        // Se houver erro, exibe a mensagem e sai da função.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", LIVROS);
        return;
    }

    // A Parte 2 da função (o loop de leitura e exibição) viria após este ponto.
}
```

Fonte: Elaborado pelo Próprio Autor

Nesta etapa é feita um laço de repetição para coletar os dados salvos dentro do arquivo Venda e ser demonstrado na interface para o usuário executa a lógica de processamento e totalização do relatório. Ela opera varrendo o arquivo VENDAS para exibir o detalhe de cada transação, enquanto realiza consultas cruzadas dinâmicas.

Figura 3.14 Função Listar Venda – Parte 2

```
while ((posicao == 0) && (fread(&venda, sizeof(venda), 1, fpvenda) == 1)) {
    // --- LÓGICA DE QUEBRA DE CONTROLE (NOVO PEDIDO) ---
    // Verifica se o item lido pertence a um novo pedido de venda (código diferente do anterior).
    if (venda.codPedido != ctrlPedido) {
        // Se for um novo pedido, o valor total do anterior é resetado.
        valorTotal = 0;
        // As funções de busca são usadas para obter o nome do livro e do cliente
        livro = localizarLivro(venda.codLivro);
        cliente = LocalizarCliente(venda.codCliente);
        printf("\n Numero do Pedido: %i", venda.codPedido);
        printf("\n Nome do Cliente: %s --- %s", venda.codCliente, cliente.nome);
        printf("\n Codigo \t Livro \t Preco \t Quantidade");
    }
    // --- LÓGICA DE VARREDURA DOS ITENS ---
    // Abertura redundante/incorreta: Tenta reabrir o arquivo VENDAS.
    if ((fpctrlLivro = fopen(VENDAS, "rb")) == NULL) {
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", VENDAS);
        return; // Se falhar, retorna prematuramente, fechando o loop externo.
    }
    // Loop interno (com lógica falha para processamento sequencial):
    while ((posicao == 0) && (fread(&ctrlLivro, sizeof(ctrlLivro), 1, fpctrlLivro) == 1)) {
        // Tenta encontrar todos os itens que pertencem ao pedido atual.
        if (venda.codPedido == ctrlLivro.codPedido) {
            // Localiza o livro para pegar o nome e preço.
            livro = localizarLivro(ctrlLivro.codLivro);
            printf("\n %i \t %s \t %.2f \t %i", ctrlLivro.codLivro, livro.nome, livro.preco, ctrlLivro.quantidade);
            // Acumula o valor na totalização.
            valorTotal = valorTotal + (ctrlLivro.quantidade * livro.preco);
        }
    }
    // Exibe o total do pedido (aqui ou na próxima quebra de controle).
    printf("\n \t Valor Total: R$ %.2f", valorTotal);
    fclose(fpctrlLivro);
    // Salva o código do pedido atual para a próxima checagem do loop principal.
    ctrlPedido = venda.codPedido;
}
```

Fonte: Elaborado pelo Próprio Autor

Nesta primeira parte da função é realizada toda declaração de *struct* e informações usadas e validações de abertura dos arquivos, inicia o processo de conclusão da transação, que envolve a geração de um relatório de fechamento e a crucial atualização do inventário.

Figura 3.15 Função Fechar Pedido – Parte 1

```c
// Define a função FecharPedido. Esta é uma função crítica de processamento que
// gera um relatório de fechamento, totaliza a venda e, o mais importante,
// realiza a baixa (atualização) do estoque.
void FecharPedido() {
    // Declaração dos ponteiros para os 3 arquivos necessários.
    FILE *fpvenda, *fplivro, *fpcliente;
    // Declaração das structs para manipular os dados lidos.
    struct regVenda venda;
    struct regLivro livro;
    struct regCliente cliente;

    // Variáveis de controle para o código do pedido, flag de busca e totalização.
    int codigoVenda, posicao = 0;
    float valorTotal, valorTotalItem = 0;

    // --- 1. ABERTURA SEGURA DO ARQUIVO VENDAS ---
    // Tenta abrir o arquivo VENDAS ("vendas.dat") em modo de Leitura Binária ("rb").
    if (((fpvenda = fopen(VENDAS, "rb")) == NULL)) {
        printf("\n Erro ao abrir arquivo %s\n", VENDAS);
        return; // Sai em caso de erro.
    }
    // --- 2. ABERTURA SEGURA DO ARQUIVO CLIENTES ---
    // Tenta abrir o arquivo CLIENTES ("clientes.dat") em modo de Leitura Binária ("rb").
    if (((fpcliente = fopen(CLIENTES, "rb")) == NULL)) {
        printf("\n Erro ao abrir arquivo %s\n", CLIENTES);
        return; // Sai em caso de erro.
    }
    // --- 3. ABERTURA CRÍTICA DO ARQUIVO LIVROS ---
    // Tenta abrir o arquivo LIVROS ("livros.dat") em modo de Leitura/Escrita Binária ("rb+").
    if (((fplivro = fopen(LIVROS, "rb+")) == NULL)) {
        printf("\n Erro ao abrir arquivo %s\n", LIVROS);
        return; // Sai em caso de erro.
    }
}
```

Fonte: Elaborado pelo Próprio Autor

A função Fechar Pedido Figura 3.15 e Figura 3.16 recebe o código da venda a ser fechada, verifica sua existência, gera o relatório e, em seguida, executa o *loop* que processa cada item para totalização e, mais importante, para a baixa no estoque.

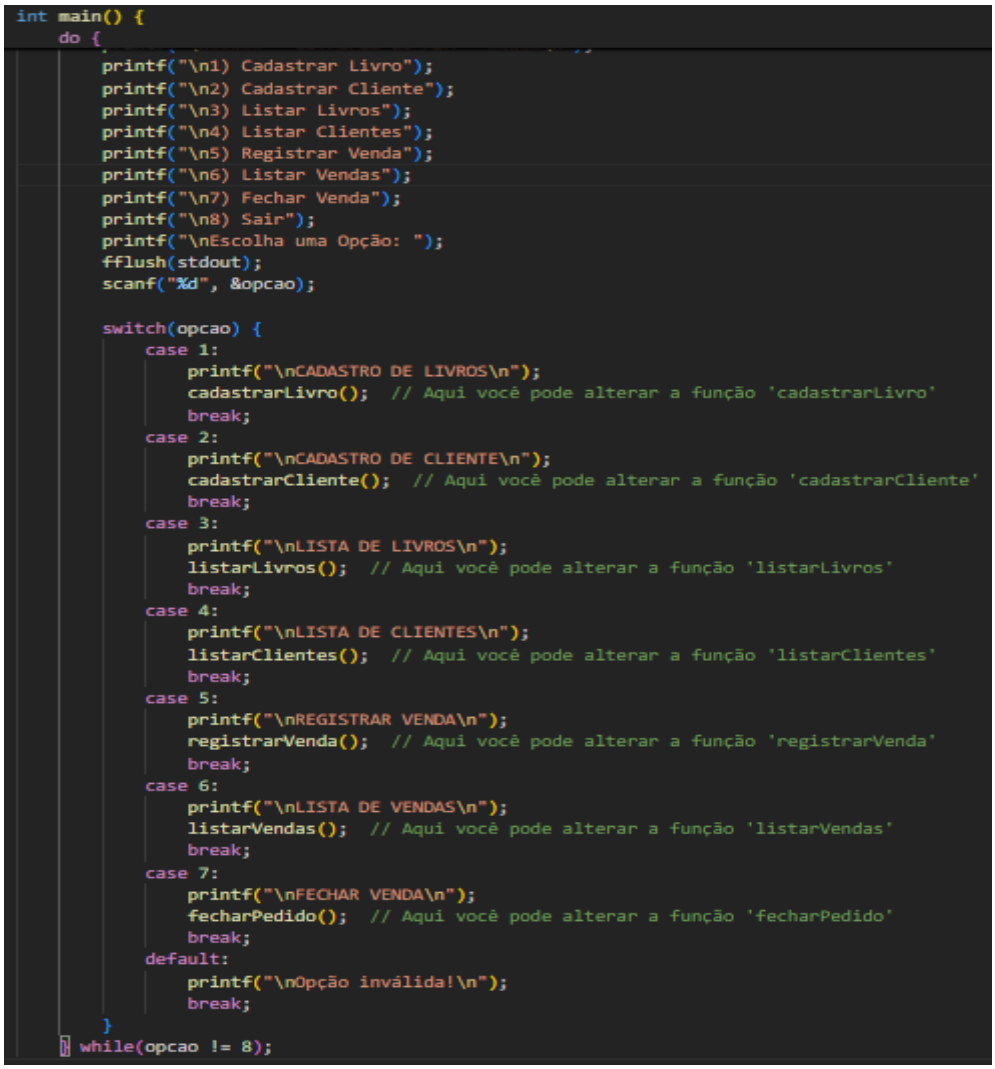
Figura 3.16 Função Fechar Pedido – Parte 2

```
// --- 4. COLETA DO CÓDIGO E VALIDAÇÃO ---
printf("\n Insira o codigo da venda que sera fechada: \n");
fflush(stdin); scanf("%i", &codigoVenda);
// Verifica se a venda existe antes de prosseguir com a busca (otimização).
if (vendaExiste(codigoVenda) == 1) {
    // Localiza o registro da venda e do cliente (para exibição do cabeçalho).
    venda = LocalizarVenda(codigoVenda);
    cliente = LocalizarCliente(venda.codCliente);
    printf("\n PEDIDO NRO: %i", venda.codPedido);
    printf("\n Cliente: %i - %s", venda.codCliente, cliente.nome);
    printf("\n \n Codigo \t Livro \t Preco \t Qtdade \t Valor Total Item");

    // --- 5. LOOP DE PROCESSAMENTO E BAIXA DE ESTOQUE ---
    rewind(fpvenda); // (Assumido que o arquivo deve ser rebobinado aqui, pois LocalizarVenda o moveu).
    // Loop principal: Varre o arquivo VENDAS em busca dos itens do pedido.
    while ((posicao == 0) && (fread(&venda, sizeof(venda), 1, fpvenda) == 1)) {
        livro = LocalizarLivro(venda.codLivro);
        if (venda.codPedido == codigoVenda) {
            valorTotalItem = venda.quantidade * livro.preco;
            printf("\n %i \t %s \t %.2f \t %i \t %.2f", venda.codLivro, livro.nome, livro.preco, venda.quantidade, valorTotalItem);
            // Acumula o valor na variável de totalização.
            valorTotal = valorTotal + (venda.quantidade * livro.preco);
            // LÓGICA DE BAIXA NO ESTOQUE (NA MEMÓRIA)
            livro.estoque = livro.estoque - venda.quantidade;
            atualizaEstoque(livro.codigo, livro.estoque);
        }
    }
    // Imprime o valor total da transação.
    printf("\n \n\t Valor total: R$ %.2f", valorTotal);
    printf("\n ---------- PEDIDO FECHADO! ----------\n");
} else {
    // Bloco executado se vendaExiste() retornar 0 (pedido não encontrado).
    printf("\n Pedido de venda nao encontrado");
    return; // Sai da função.
}
// Fecha todos os arquivos abertos na Parte 1, liberando recursos.
fclose(fpvenda);
fclose(fpcliente);
fclose(fplivro);
```

Fonte: Elaborado pelo Próprio Autor

A Figura 3.17 Menu de Interface é a construção do menu onde no qual o usuário visualiza em sua tela, também é o caminho que permite a aplicação acessar as funções para manipulação dos dados. A arquitetura é construída em torno de três entidades principais, definidas por *structs*, e funções especializadas que as manipulam. O desenvolvimento dos códigos até este ponto ocorreu em colaboração com o docente responsável pela matéria. A implementação apresentada na Figura 3.18, juntamente com as demais etapas, constitui a atividade de avaliação escolhida para integrar o presente portfólio. O conteúdo programático enfoca a aplicação de rotinas para registro e fechamento de pedidos de venda, além da elaboração de um relatório de controle de Estoque Mínimo.

Figura 3.17 Menu de Interface

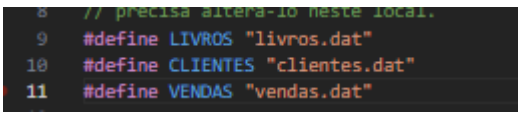


```c
int main() {
    do {
        printf("\n1) Cadastrar Livro");
        printf("\n2) Cadastrar Cliente");
        printf("\n3) Listar Livros");
        printf("\n4) Listar Clientes");
        printf("\n5) Registrar Venda");
        printf("\n6) Listar Vendas");
        printf("\n7) Fechar Venda");
        printf("\n8) Sair");
        printf("\nEscolha uma Opção: ");
        fflush(stdout);
        scanf("%d", &opcao);

        switch(opcao) {
            case 1:
                printf("\nCADASTRO DE LIVROS\n");
                cadastrarLivro();  // Aqui você pode alterar a função 'cadastrarLivro'
                break;
            case 2:
                printf("\nCADASTRO DE CLIENTE\n");
                cadastrarCliente();  // Aqui você pode alterar a função 'cadastrarCliente'
                break;
            case 3:
                printf("\nLISTA DE LIVROS\n");
                listarLivros();  // Aqui você pode alterar a função 'listarLivros'
                break;
            case 4:
                printf("\nLISTA DE CLIENTES\n");
                listarClientes();  // Aqui você pode alterar a função 'listarClientes'
                break;
            case 5:
                printf("\nREGISTRAR VENDA\n");
                registrarVenda();  // Aqui você pode alterar a função 'registrarVenda'
                break;
            case 6:
                printf("\nLISTA DE VENDAS\n");
                listarVendas();  // Aqui você pode alterar a função 'listarVendas'
                break;
            case 7:
                printf("\nFECHAR VENDA\n");
                fecharPedido();  // Aqui você pode alterar a função 'fecharPedido'
                break;
            default:
                printf("\nOpção inválida!\n");
                break;
        }
    } while(opcao != 8);
```

Fonte: Elaborado pelo Próprio Autor

A Figura 3.18 é responsável pela criação de um constante chamada “VENDAS” de forma que sempre que for chamada representará “`vendas.dat`”.

Figura 3.18 Criação da define VENDAS



```c
// precisa altera-lo neste local.
#define LIVROS "livros.dat"
#define CLIENTES "clientes.dat"
#define VENDAS "vendas.dat"
```

Fonte: Elaborado pelo Próprio Autor

Neste bloco de código cria-se uma *struct* chamada “`regVenda`” para ser utilizada na função de registro de venda.

Figura 3.19 Criação da *struct* regVenda

```
// Define uma estrutura para representar uma venda ou um pedido.
// Esta                                    a um livro e a uma quantidade vendida.
struct r   int regVenda::codPedido
    int codPedido;    // Código único do pedido ou da venda.
    int codCliente;   // Código do cliente que realizou a compra.
    int codLivro;     // Código do livro que foi vendido.
    int quantidade;   // Quantidade de unidades do livro vendidas neste pedido.
};
```

Fonte: Elaborado pelo Próprio Autor

A Figura 3.20 refere-se à função `localizarVenda`, o qual tem por objetivo acessar a *struct* `regVenda` e realizar a confirmação da existência da venda.

Figura 3.20 Função localizarVenda

```
196 struct regVenda LocalizarVenda(int codVenda) {
197     // Declara um ponteiro de arquivo para manipulação do arquivo de vendas.
198     FILE *fpvenda;
199     // Declara uma struct para armazenar os dados do registro lido do arquivo.
200     struct regVenda venda;
201     // 'posicao' é um flag: 0 (não encontrado) ou 1 (encontrado).
202     int posicao = 0;
203
204     // Abre o arquivo VENDAS ("vendas.dat") em modo de Leitura Binária ("rb").
205     if((fpvenda = fopen(VENDAS, "rb")) == NULL) {
206         // Verifica se houve ERRO ao abrir o arquivo. Se sim, imprime uma mensagem.
207         printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", VENDAS);
208     }
209     // Inicializa a flag novamente antes do loop (apesar de já ser 0, reforça a intenção).
210     posicao = 0;
211     // Loop de busca sequencial: Continua enquanto a venda não for encontrada (posicao == 0)
212     // E a leitura de um novo registro for bem-sucedida (fread() == 1).
213     // Nota: O loop é interrompido imediatamente após o primeiro registro correspondente.
214     while((posicao == 0) && (fread(&venda, sizeof(venda), 1, fpvenda) == 1)) {
215         // Compara o código do pedido lido com o código procurado.
216         if(venda.codPedido == codVenda) {
217             // Se houver correspondência, muda a flag para 1, sinalizando sucesso na busca.
218             posicao = 1;
219         }
220     }
221     // Fecha o arquivo. Essencial para liberar o recurso.
222     fclose(fpvenda);
223     // Bloco de tratamento de erro e sinalização:
224     // Se a flag for 0, significa que o registro não foi localizado após a varredura.
225     if(posicao == 0) {
226         // Define o campo chave com -1 para sinalizar a falha da busca.
227         venda.codPedido = -1;
228     }
229     // Retorna a struct de venda (encontrada ou com o código de falha -1).
230     return venda;
231 }
```

Fonte: Elaborado pelo Próprio Autor

A função denominada "`registrarVenda`", apresentada na Figura 3.21, tem como propósito efetuar o registro de uma nova transação. Nesta seção do código, são criadas as *structs* necessárias, é realizada a coleta de dados e, por último, a verificação da existência do cliente.

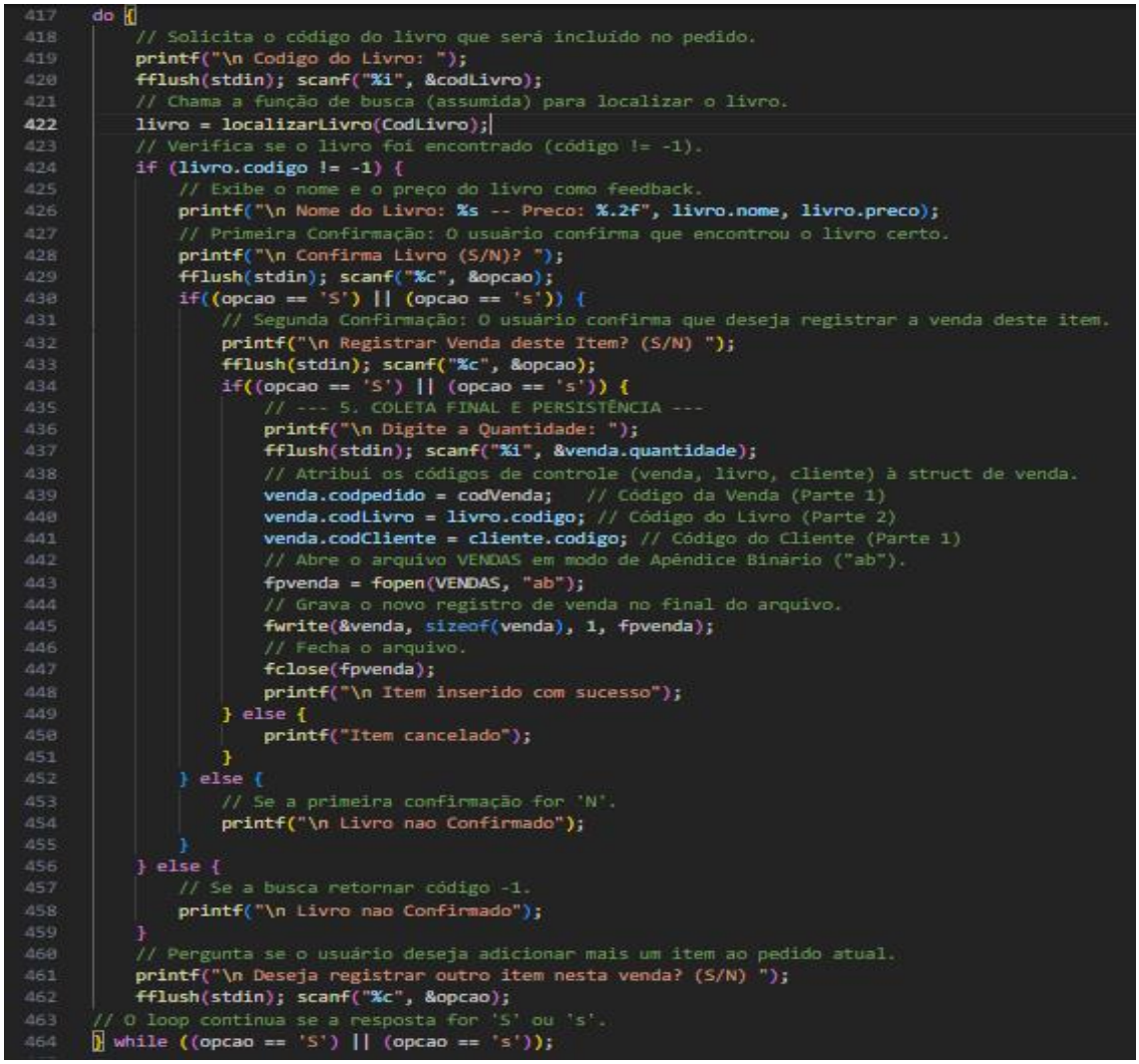
Figura 3.21 Função registrarVenda Parte 1

```c
void registrarVenda() {
    // Declaração de ponteiros para acesso aos 3 arquivos necessários.
    FILE *fplivro, *fpcliente, *fpvenda;
    // Declaração das structs que serão usadas para manipular os dados.
    struct regLivro livro;
    struct regCliente cliente;
    struct regVenda venda;
    // Variável para armazenar a opção do usuário (S/N).
    char opcao;
    int posicao, codVenda, codCliente, codLivro;
    // --- 1. COLETA DE DADOS INICIAIS ---
    // Solicita o código da nova venda.
    printf("\n Codigo Venda: ");
    fflush(stdin); scanf("%i", &codVenda);
    // Solicita o código do cliente para a validação.
    printf("\n Codigo do Cliente: ");
    fflush(stdin); scanf("%i", &codCliente);
    // --- 2. VALIDAÇÃO DO CLIENTE ---
    // Tenta localizar o cliente usando o código fornecido.
    cliente = LocalizarCliente(codCliente);
    // Verifica se a função LocalizarCliente retornou o código de erro (-1).
    if(cliente.codigo == -1) {
        printf("\n Nao foi possivel localizar Cliente");
        return; // Cancela a operação se o cliente não for encontrado.
    } else {
        // Se o cliente foi encontrado, exibe seu nome como feedback.
        printf("\n Cliente Localizado: %s", cliente.nome);
    }
    // --- 3. CONFIRMAÇÃO DA OPERAÇÃO ---
    printf("\n Confirma Cliente? (S/N): ");
    fflush(stdin); scanf("%c", &opcao);
    // Verifica se a operação foi cancelada.
    if(((opcao == 'N') || (opcao == 'n'))) {
        printf("\n Operacao Cancelada com Sucesso");
        return; // Sai da função.
    }
    // A Parte 2 da função (busca do livro e gravação) viria após este ponto.
}
```

Fonte: Elaborado pelo Próprio Autor

A Figura 3.22 ilustra a continuidade da função “`registrarVenda`”, que começou na Figura 3.21. Nesta parte do código, a coleta de dados para a venda é finalizada e as etapas para o registro propriamente dito são iniciadas.

Figura 3.22 Função registrarVenda Parte 2

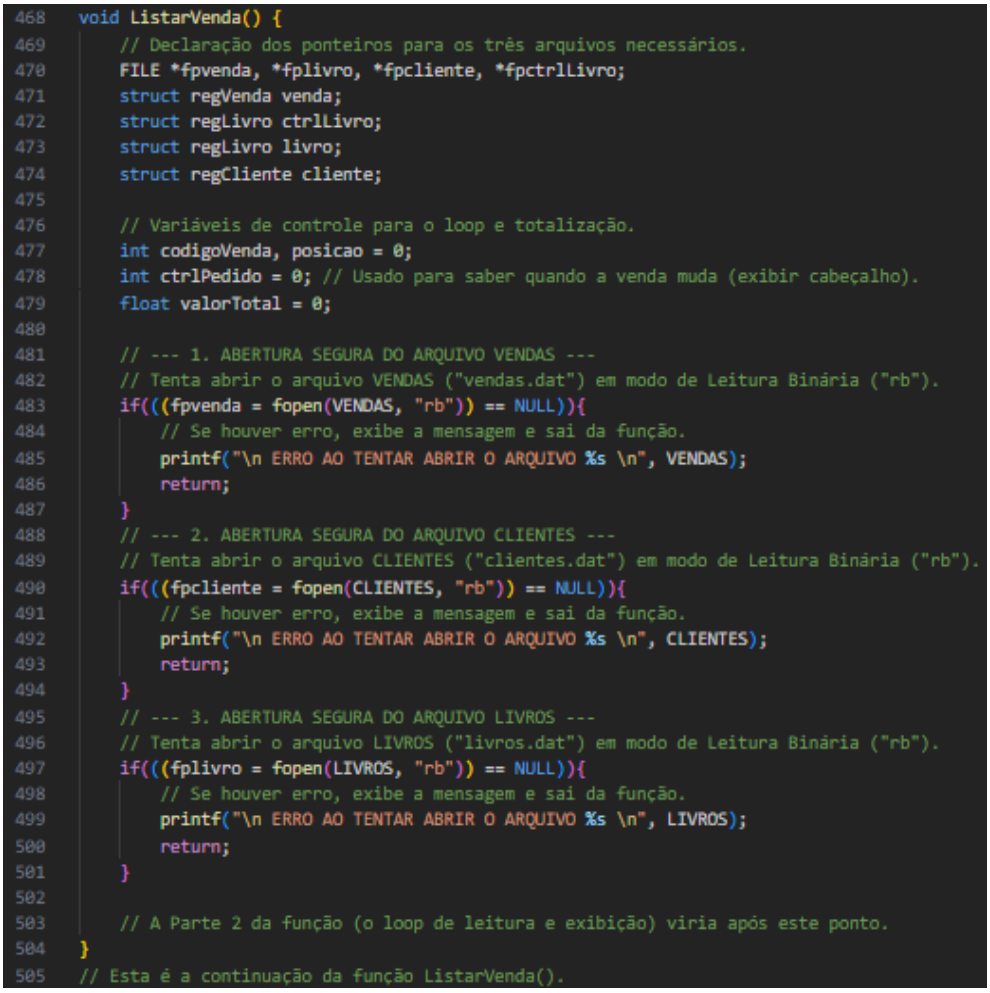


```
417  do {
418      // Solicita o código do livro que será incluído no pedido.
419      printf("\n Codigo do Livro: ");
420      fflush(stdin); scanf("%i", &codLivro);
421      // Chama a função de busca (assumida) para localizar o livro.
422      livro = localizarLivro(CodLivro);
423      // Verifica se o livro foi encontrado (código != -1).
424      if (livro.codigo != -1) {
425          // Exibe o nome e o preço do livro como feedback.
426          printf("\n Nome do Livro: %s -- Preco: %.2f", livro.nome, livro.preco);
427          // Primeira Confirmação: O usuário confirma que encontrou o livro certo.
428          printf("\n Confirma Livro (S/N)? ");
429          fflush(stdin); scanf("%c", &opcao);
430          if((opcao == 'S') || (opcao == 's')) {
431              // Segunda Confirmação: O usuário confirma que deseja registrar a venda deste item.
432              printf("\n Registrar Venda deste Item? (S/N) ");
433              fflush(stdin); scanf("%c", &opcao);
434              if((opcao == 'S') || (opcao == 's')) {
435                  // --- 5. COLETA FINAL E PERSISTÊNCIA ---
436                  printf("\n Digite a Quantidade: ");
437                  fflush(stdin); scanf("%i", &venda.quantidade);
438                  // Atribui os códigos de controle (venda, livro, cliente) à struct de venda.
439                  venda.codpedido = codVenda;   // Código da Venda (Parte 1)
440                  venda.codLivro = livro.codigo; // Código do Livro (Parte 2)
441                  venda.codCliente = cliente.codigo; // Código do Cliente (Parte 1)
442                  // Abre o arquivo VENDAS em modo de Apêndice Binário ("ab").
443                  fpvenda = fopen(VENDAS, "ab");
444                  // Grava o novo registro de venda no final do arquivo.
445                  fwrite(&venda, sizeof(venda), 1, fpvenda);
446                  // Fecha o arquivo.
447                  fclose(fpvenda);
448                  printf("\n Item inserido com sucesso");
449              } else {
450                  printf("Item cancelado");
451              }
452          } else {
453              // Se a primeira confirmação for 'N'.
454              printf("\n Livro nao Confirmado");
455          }
456      } else {
457          // Se a busca retornar código -1.
458          printf("\n Livro nao Confirmado");
459      }
460      // Pergunta se o usuário deseja adicionar mais um item ao pedido atual.
461      printf("\n Deseja registrar outro item nesta venda? (S/N) ");
462      fflush(stdin); scanf("%c", &opcao);
463  // O loop continua se a resposta for 'S' ou 's'.
464  } while ((opcao == 'S') || (opcao == 's'));
```

Fonte: Elaborado pelo Próprio Autor

Ilustrada nas Figura 3.23 e Figura 3.24, a função `listarVenda` é responsável por recuperar e listar as vendas contidas no arquivo de dados `vendas.dat`.

Figura 3.23 Função listarVenda parte 1



```
void ListarVenda() {
    // Declaração dos ponteiros para os três arquivos necessários.
    FILE *fpvenda, *fplivro, *fpcliente, *fpctrlLivro;
    struct regVenda venda;
    struct regLivro ctrlLivro;
    struct regLivro livro;
    struct regCliente cliente;

    // Variáveis de controle para o loop e totalização.
    int codigoVenda, posicao = 0;
    int ctrlPedido = 0; // Usado para saber quando a venda muda (exibir cabeçalho).
    float valorTotal = 0;

    // --- 1. ABERTURA SEGURA DO ARQUIVO VENDAS ---
    // Tenta abrir o arquivo VENDAS ("vendas.dat") em modo de Leitura Binária ("rb").
    if(((fpvenda = fopen(VENDAS, "rb")) == NULL)){
        // Se houver erro, exibe a mensagem e sai da função.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", VENDAS);
        return;
    }
    // --- 2. ABERTURA SEGURA DO ARQUIVO CLIENTES ---
    // Tenta abrir o arquivo CLIENTES ("clientes.dat") em modo de Leitura Binária ("rb").
    if(((fpcliente = fopen(CLIENTES, "rb")) == NULL)){
        // Se houver erro, exibe a mensagem e sai da função.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", CLIENTES);
        return;
    }
    // --- 3. ABERTURA SEGURA DO ARQUIVO LIVROS ---
    // Tenta abrir o arquivo LIVROS ("livros.dat") em modo de Leitura Binária ("rb").
    if(((fplivro = fopen(LIVROS, "rb")) == NULL)){
        // Se houver erro, exibe a mensagem e sai da função.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", LIVROS);
        return;
    }

    // A Parte 2 da função (o loop de leitura e exibição) viria após este ponto.
}
// Esta é a continuação da função ListarVenda().
```

Fonte: Elaborado pelo Próprio Autor

Figura 3.24 Função listarVenda parte 2

```
// Esta é a continuação da função ListarVenda().

while ((posicao == 0) && (fread(&venda, sizeof(venda), 1, fpvenda) == 1)) {
    // --- LÓGICA DE QUEBRA DE CONTROLE (NOVO PEDIDO) ---
    // Verifica se o item lido pertence a um novo pedido de venda (código diferente do anterior).
    if (venda.codPedido != ctrlPedido) {
        // Se for um novo pedido, o valor total do anterior é resetado.
        valorTotal = 0;
        // As funções de busca são usadas para obter o nome do livro e do cliente
        livro = localizarLivro(venda.codLivro);
        cliente = LocalizarCliente(venda.codCliente);
        printf("\n Numero do Pedido: %i", venda.codPedido);
        printf("\n Nome do Cliente: %s --- %s", venda.codCliente, cliente.nome);
        printf("\n Codigo \t Livro \t Preco \t Quantidade");
    }
    // --- LÓGICA DE VARREDURA DOS ITENS ---
    // Abertura redundante/incorreta: Tenta reabrir o arquivo VENDAS.
    if ((fpctrlLivro = fopen(VENDAS, "rb")) == NULL) {
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s \n", VENDAS);
        return; // Se falhar, retorna prematuramente, fechando o loop externo.
    }
    // Loop interno (com lógica falha para processamento sequencial):
    while ((posicao == 0) && (fread(&ctrlLivro, sizeof(ctrlLivro), 1, fpctrlLivro) == 1)) {
        // Tenta encontrar todos os itens que pertencem ao pedido atual.
        if (venda.codPedido == ctrlLivro.codPedido) {
            // Localiza o livro para pegar o nome e preço.
            livro = localizarLivro(ctrlLivro.codLivro);
            printf("\n %i \t %s \t %.2f \t %i", ctrlLivro.codLivro, livro.nome, livro.preco, ctrlLivro.quantidade);
            // Acumula o valor na totalização.
            valorTotal = valorTotal + (ctrlLivro.quantidade * livro.preco);
        }
    }
    // Exibe o total do pedido (aqui ou na próxima quebra de controle).
    printf("\n \t Valor Total: R$ %.2f", valorTotal);
    fclose(fpctrlLivro);
    // Salva o código do pedido atual para a próxima checagem do loop principal.
    ctrlPedido = venda.codPedido;
}
// --- ENCERRAMENTO ---
```

Fonte: Elaborado pelo Próprio Autor

A função “`atualizarEstoque`”, ilustrada na Figura 3.25, é um recurso de suporte cuja principal utilidade é modificar o inventário sempre que acionada. Quando invocada, ela requer dois dados de entrada: o código do item e a quantidade a ser utilizada na atualização.

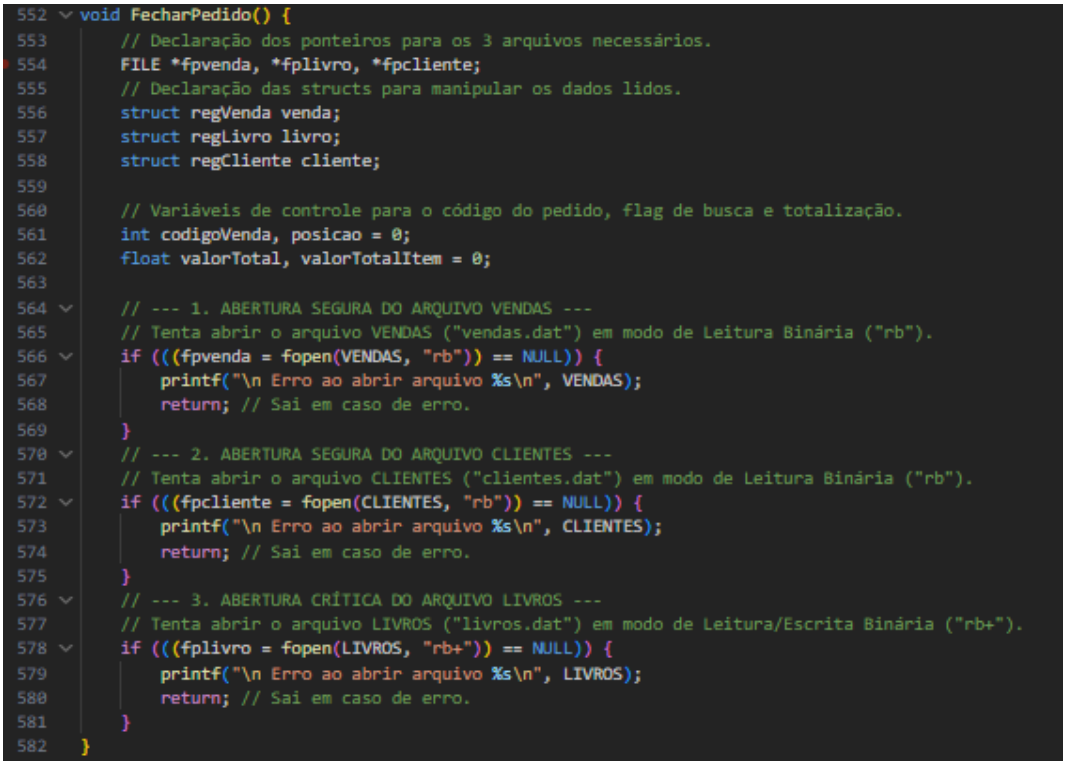
Figura 3.25 Função atualizaEstoque

```c
void atualizaEstoque(int codItemAtt, int quantidadeEst) {
    // Declara o ponteiro de arquivo para manipulação.
    FILE *fplivro;
    // Declara a struct para armazenar o registro do livro lido/modificado.
    struct regLivro livro;
    // Flag de controle: 0 (não encontrado) ou 1 (encontrado).
    int posicao = 0;
    // Abre o arquivo LIVROS ("livros.dat") em modo de Leitura/Escrita Binária ("rb+").
    // O modo "rb+" permite tanto ler quanto escrever em qualquer posição do arquivo.
    if (((fplivro = fopen(LIVROS, "rb+")) == NULL)) {
        // Verifica erro na abertura. Se falhar, exibe mensagem e sai da função.
        printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s\n", LIVROS);
        return;
    }
    // Continua enquanto o registro não for encontrado (posicao == 0)
    // E houver registros para ler (fread retorna 1).
    while ((posicao == 0) && (fread(&livro, sizeof(livro), 1, fplivro) == 1)) {
        // Compara o código do livro lido com o código procurado.
        if (livro.codigo == codItemAtt) {
            // Se encontrar, define a flag como 1 (encontrado).
            posicao = 1;
        }
    }
    // Verifica se a busca falhou (posicao == 0).
    if (posicao == 0) {
        printf("\n NAO FOI POSSIVEL LOCALIZAR LIVRO ");
        fclose(fplivro); // Fecha o arquivo antes de sair.
        return;          // Encerra a função.
    }
    // Atualiza o campo 'estoque' da struct que está na memória com o novo valor.
    livro.estoque = quantidadeEst;
    // fseek move o ponteiro 'sizeof(livro)' bytes PARA TRÁS,
    // a partir da posição atual (que é o FIM do registro lido).
    fseek(fplivro, -sizeof(livro), 1);
    // Grava a struct 'livro' (agora modificada) sobre o registro antigo.
    fwrite(&livro, sizeof(livro), 1, fplivro);
    // Fecha o arquivo, garantindo que a alteração seja salva no disco.
    fclose(fplivro);
}
```

Fonte: Elaborado pelo Próprio Autor

As Figura 3.26 e Figura 3.27 exibem a função `FecharPedido`, que tem a responsabilidade de concluir a transação de venda registrada no arquivo "`vendas.dat`", simultaneamente, realizar a atualização do estoque.

Figura 3.26 FecharPedido parte 1



```c
void FecharPedido() {
    // Declaração dos ponteiros para os 3 arquivos necessários.
    FILE *fpvenda, *fplivro, *fpcliente;
    // Declaração das structs para manipular os dados lidos.
    struct regVenda venda;
    struct regLivro livro;
    struct regCliente cliente;

    // Variáveis de controle para o código do pedido, flag de busca e totalização.
    int codigoVenda, posicao = 0;
    float valorTotal, valorTotalItem = 0;

    // --- 1. ABERTURA SEGURA DO ARQUIVO VENDAS ---
    // Tenta abrir o arquivo VENDAS ("vendas.dat") em modo de Leitura Binária ("rb").
    if (((fpvenda = fopen(VENDAS, "rb")) == NULL)) {
        printf("\n Erro ao abrir arquivo %s\n", VENDAS);
        return; // Sai em caso de erro.
    }
    // --- 2. ABERTURA SEGURA DO ARQUIVO CLIENTES ---
    // Tenta abrir o arquivo CLIENTES ("clientes.dat") em modo de Leitura Binária ("rb").
    if (((fpcliente = fopen(CLIENTES, "rb")) == NULL)) {
        printf("\n Erro ao abrir arquivo %s\n", CLIENTES);
        return; // Sai em caso de erro.
    }
    // --- 3. ABERTURA CRÍTICA DO ARQUIVO LIVROS ---
    // Tenta abrir o arquivo LIVROS ("livros.dat") em modo de Leitura/Escrita Binária ("rb+").
    if (((fplivro = fopen(LIVROS, "rb+")) == NULL)) {
        printf("\n Erro ao abrir arquivo %s\n", LIVROS);
        return; // Sai em caso de erro.
    }
}
```

Fonte: Elaborado pelo Próprio Autor

Figura 3.27 FecharPedido parte 2

```c
// Esta é a Parte 2 (e o encerramento) da função FecharPedido().

    // --- 4. COLETA DO CÓDIGO E VALIDAÇÃO ---
    printf("\n Insira o codigo da venda que sera fechada: \n");
    fflush(stdin); scanf("%i", &codigoVenda);
    // Verifica se a venda existe antes de prosseguir com a busca (otimização).
    if (vendaExiste(codigoVenda) == 1) {
        // Localiza o registro da venda e do cliente (para exibição do cabeçalho).
        venda = LocalizarVenda(codigoVenda);
        cliente = LocalizarCliente(venda.codCliente);
        printf("\n PEDIDO NRO: %i", venda.codPedido);
        printf("\n Cliente: %i - %s", venda.codCliente, cliente.nome);
        printf("\n \n Codigo \t Livro \t Preco \t Qtdade \t Valor Total Item");

        // --- 5. LOOP DE PROCESSAMENTO E BAIXA DE ESTOQUE ---
        rewind(fpvenda); // (Assumido que o arquivo deve ser rebobinado aqui, pois LocalizarVenda o moveu).
        // Loop principal: Varre o arquivo VENDAS em busca dos itens do pedido.
        while ((posicao == 0) && (fread(&venda, sizeof(venda), 1, fpvenda) == 1)) {
            livro = LocalizarLivro(venda.codLivro);
            if (venda.codPedido == codigoVenda) {
                valorTotalItem = venda.quantidade * livro.preco;
                printf("\n %i \t %s \t %.2f \t %i \t %.2f", venda.codLivro, livro.nome, livro.preco, venda.quantidade, valorTotalItem);
                // Acumula o valor na variável de totalização.
                valorTotal = valorTotal + (venda.quantidade * livro.preco);
                // LÓGICA DE BAIXA NO ESTOQUE (NA MEMÓRIA)
                livro.estoque = livro.estoque - venda.quantidade;
                atualizaEstoque(livro.codigo, livro.estoque);
            }
        }
        // Imprime o valor total da transação.
        printf("\n \n\t Valor total: R$ %.2f", valorTotal);
        printf("\n ---------- PEDIDO FECHADO! ----------\n");
    } else {
        // Bloco executado se vendaExiste() retornar 0 (pedido não encontrado).
        printf("\n Pedido de venda nao encontrado");
        return; // Sai da função.
    }
    // Fecha todos os arquivos abertos na Parte 1, liberando recursos.
    fclose(fpvenda);
    fclose(fpcliente);
    fclose(fplivro);
```

Fonte: Elaborado pelo Próprio Autor

Demonstrada na Figura 3.28, a rotina da Função `atualizaEstoque` é encarregada de criar um relatório em formato TXT que lista os itens que possuem o número de unidades em estoque especificado.

Figura 3.28 Função atualizaEstoque

```
337 void atualizaEstoque(int codItemAtt, int quantidadeEst) {
338     // Declara o ponteiro de arquivo para manipulação.
339     FILE *fplivro;
340     // Declara a struct para armazenar o registro do livro lido/modificado.
341     struct reglivro livro;
342     // Flag de controle: 0 (não encontrado) ou 1 (encontrado).
343     int posicao = 0;
344     // Abre o arquivo LIVROS ("livros.dat") em modo de Leitura/Escrita Binária ("rb+").
345     // O modo "rb+" permite tanto ler quanto escrever em qualquer posição do arquivo.
346     if (((fplivro = fopen(LIVROS, "rb+")) == NULL)) {
347         // Verifica erro na abertura. Se falhar, exibe mensagem e sai da função.
348         printf("\n ERRO AO TENTAR ABRIR O ARQUIVO %s\n", LIVROS);
349         return;
350     }
351     // Continua enquanto o registro não for encontrado (posicao == 0)
352     // E houver registros para ler (fread retorna 1).
353     while ((posicao == 0) && (fread(&livro, sizeof(livro), 1, fplivro) == 1)) {
354         // Compara o código do livro lido com o código procurado.
355         if (livro.codigo == codItemAtt) {
356             // Se encontrar, define a flag como 1 (encontrado).
357             posicao = 1;
358         }
359     }
360     // Verifica se a busca falhou (posicao == 0).
361     if (posicao == 0) {
362         printf("\n NAO FOI POSSIVEL LOCALIZAR LIVRO ");
363         fclose(fplivro); // Fecha o arquivo antes de sair.
364         return;          // Encerra a função.
365     }
366     // Atualiza o campo 'estoque' da struct que está na memória com o novo valor.
367     livro.estoque = quantidadeEst;
368     // fseek move o ponteiro 'sizeof(livro)' bytes PARA TRÁS,
369     // a partir da posição atual (que é o FIM do registro lido).
370     fseek(fplivro, -sizeof(livro), 1);
371     // Grava a struct 'livro' (agora modificada) sobre o registro antigo.
372     fwrite(&livro, sizeof(livro), 1, fplivro);
373     // Fecha o arquivo, garantindo que a alteração seja salva no disco.
374     fclose(fplivro);
375 }
```

Fonte: Elaborado pelo Próprio Autor

3.3 Conclusão

O desenvolvimento de um sistema para livraria utilizando a linguagem C constituiu uma experiência profundamente enriquecedora. Através deste projeto, foi possível aplicar de maneira prática os conhecimentos teóricos adquiridos na disciplina correspondente, permitindo o enfrentamento de um desafio real e o desenvolvimento de soluções eficazes.

Este programa não apenas otimiza e automatiza a rotina de gerenciamento, mas também se estabelece como um marco significativo na minha trajetória de aprendizado. Ele serviu para consolidar, em um único produto, todas as técnicas e conhecimentos assimilados.

4 CIRCUITOS EM ARDUINO

A discussão deste capítulo centra-se na elaboração e montagem de três configurações de circuito Arduino: sensor de presença e sensores de proximidade (nas versões com e sem *display*). O trabalho foi desenvolvido na linguagem de programação Arduino, empregando-se múltiplas técnicas e pesquisas a fim de assegurar sua completa adequação aos parâmetros exigidos pelo tema.

4.1 DESENVOLVIMENTO

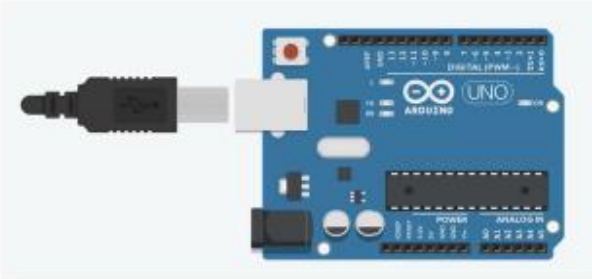
Para a realização deste estudo, a disciplina de Programação Orientada a Objetos Avançada forneceu a orientação necessária para a criação do código e a execução dos circuitos Arduino, cada qual com suas especificações detalhadas. Em qualquer projeto Arduino, a placa de prototipagem eletrônica é um componente central e indispensável. Para este trabalho, optou-se pela utilização do ambiente virtual Tinkercad, acessível em Tinkercad.com, para a simulação dos componentes. Os circuitos aqui apresentados foram montados virtualmente utilizando a placa Arduino Uno R3 e uma Protoboard (placa de circuito experimental), que atua como um expansor da placa principal.

“O aprendizado de lógica de programação e manipulação de eventos em projetos práticos fornece suporte conceitual importante para o desenvolvimento de códigos”. Freeman (2016), em *Use a Cabeça!: Programação JavaScript*, forneceu suporte conceitual importante para o desenvolvimento do código Arduino, auxiliando na estruturação de comandos, ciclos e respostas a sensores.

O primeiro circuito consiste na integração de um sensor de movimento, um *buzzer* e um LED. A exigência específica para esta atividade é que, ao ser detectado um movimento, o *buzzer* (componente emissor de som) e o LED sejam ativados simultaneamente, seguindo um ciclo de 1 segundo ligado e 1 segundo desligado.

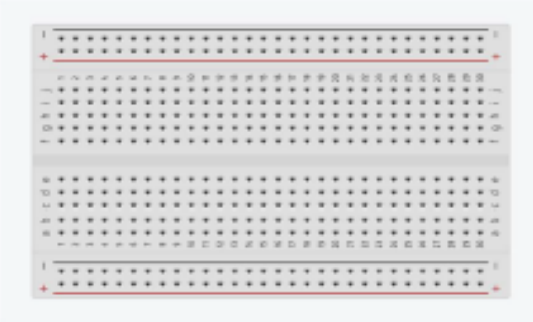
As imagens dos componentes apresentados neste trabalho são provenientes do ambiente virtual, mas seu design replica fielmente os componentes físicos existentes, a saber:

Figura 4.1 Arduino Uno R3 (Placa de Prototipagem Eletrônica):



Fonte: Captura de tela do software Tinkercad (2025)

Figura 4.2 Protoboard ou Breadboard (Placa perfurada para prototipagem rápida)

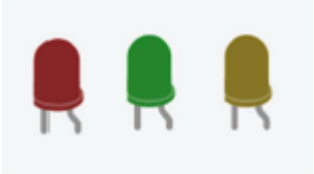


Fonte: Captura de tela do software Tinkercad (2025)

Figura 4.3 Jumpers – Fios coloridos para interligar os componentes.

Fonte Captura de tela do software Tinkercad (2025)

Figura 4.4 LEDs

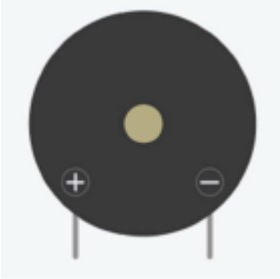


Fonte: Captura de tela do software Tinkercad (2025)

Figura 4.5 Resistores – Limitadores de corrente elétrica

Fonte: Captura de tela do software Tinkercad (2025)

Figura 4.6 Buzzer – Emissor de Som



Fonte: Captura de tela do software Tinkercad (2025)

Figura 4.7 Sensor de Movimento

Fonte: Captura de tela do software Tinkercad (2025)

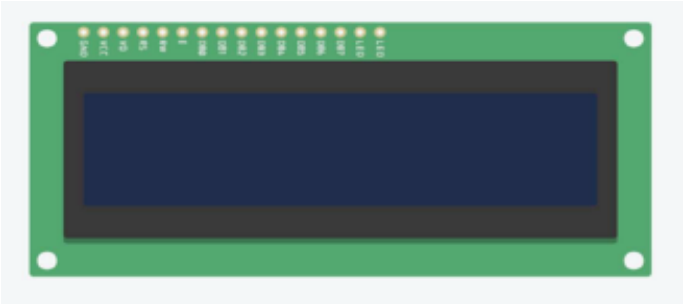
Figura 4.8 Sensor de Proximidade

Fonte: Captura de tela do software Tinkercad (2025)

Figura 4.9 Potenciômetro

Fonte: Captura de tela do software Tinkercad (2025)

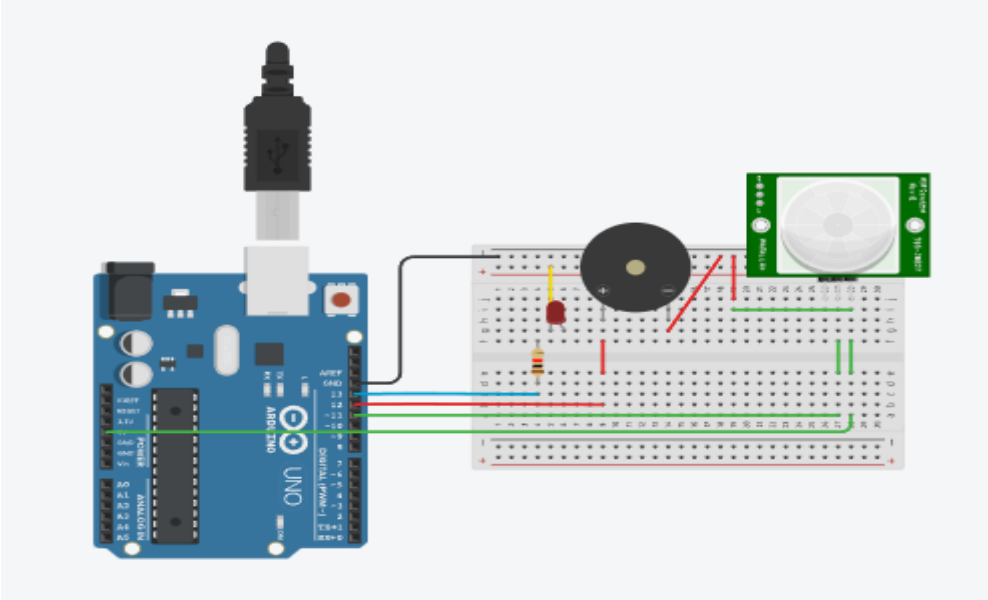
Figura 4.10 Display de LED



Fonte: Captura de tela do software Tinkercad (2025)

Na figura 4.11 pode-se verificar como fica a estrutura do circuito montada que foi disponibilizado para desenvolvimento do código de programação.

Figura 4.11 Circuito Sensor de Movimento



Fonte: Autoria própria via software Tinkercad

O começo do desenvolvimento do código é exibido na Figura 4.12. Nesta etapa, é realizada a declaração das variáveis para identificar as portas da placa de prototipagem às quais os componentes estão conectados. De acordo com o circuito (apresentado na Figura 4.11), o LED utiliza a porta 13, o sensor a porta 11 e a buzina (*Buzzer*) a porta 12.

Figura 4.12 Declaração de variáveis Circuito 1

```
1
2  // C++ code
3  //
4  int led = 13;
5  int sensor = 11;
6  int buzina = 12;
7
```

Fonte: Elaborado pelo Próprio Autor

Figura 4.13 Configuração de Output e Input circuito 1

```
9
10 void setup()
11 {
12    pinMode(led, OUTPUT);
13   pinMode(sensor, INPUT);
14   pinMode(buzina, OUTPUT);
15 }
16
17 void loop()
```

Fonte: Elaborado pelo Próprio Autor

Esta etapa finaliza o código, definindo o comportamento contínuo dos componentes. Tudo o que está inserido em *void loop* é executado de forma ininterrupta. A Figura 4.14 detalha que, ao identificar movimento, o sensor envia um valor 2. Este valor, por sua vez, dispara a ativação cíclica do *buzzer* e do LED. Essa sequência (2 segundos ligados, 2 segundos desligados) perdura até que o sensor não detecte mais presença e o valor de leitura retorne a 0.

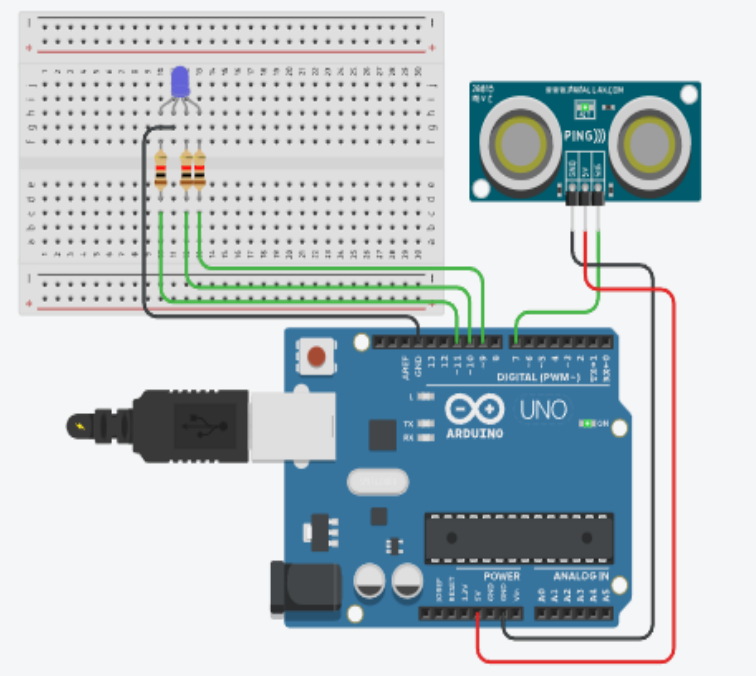
Figura 4.14 Construção do código infinito.

```
void loop()
{
   if (digitalRead(sensor) == 1) {
   digitalWrite(led, 1);
   digitalWrite(buzina, 1);
   delay(2000);
   digitalWrite(led, 0);
   digitalWrite(buzina, 0);
   delay(2000);
   digitalWrite(led, 1);
   digitalWrite(buzina, 1);
   delay(2000);
   digitalWrite(led, 0);
   digitalWrite(buzina, 0);
   delay(2000);
   digitalWrite(led, 1);
   digitalWrite(buzina, 1);
   delay(2000);
   digitalWrite(led, 0);
   digitalWrite(buzina, 0);
   delay(2000);
 }
}
```

Fonte: Elaborado pelo Próprio Autor

Na figura 4.15 pode-se verificar a estrutura de um circuito de Sensor de Proximidade, que foi montada e disponibilizado para desenvolvimento do código de programação

Figura 4.15 Circuito Sensor de Proximidade Circuito 2



Fonte: Autoria própria via software Tinkercad

O código define as portas digitais do Arduino que serão utilizadas. A variável pingPin é definida como o pino 7 para a comunicação com o sensor ultrassônico. Além disso, são declaradas três variáveis (red, blue, green) para controlar um sistema de sinalização visual (LEDs), conectadas aos pinos 11, 10 e 9, respectivamente. A função *setup*() configura os pinos digitais. Os pinos red (11), blue (10) e green (9) são configurados como saídas (*OUTPUT*) para que o Arduino possa enviar sinal para ligar ou desligar os LEDs. A comunicação serial é iniciada a 9600 *baud* para imprimir os valores de distância no monitor, conforme representado na Figura 4.16.

Figura 4.16 Declaração de variáveis e Configuração Inicial circuito 2

```
1  const int pingPin = 7;
2  int red=11;
3  int blue=10;
4  int green=9;
5  void setup() {
6
7    Serial.begin(9600);
8    pinMode(red,OUTPUT);
9    pinMode(blue,OUTPUT);
10   pinMode(green,OUTPUT);
11
12 }
```

Fonte: Elaborado pelo Próprio Autor

Descrição da Figura 4.17, O bloco *loop*() é responsável pela medição contínua da distância e pela lógica de sinalização. Medição: O código dispara o sensor ultrassônico e usa a função `pulseIn`() para medir o tempo que o pulso leva para retornar (`duration`). Esse tempo é convertido em distâncias em polegadas (inches) e centímetros (cm). Lógica de Sinalização: O código implementa um sistema de três zonas de distância, controlando os LEDs:

a) Distância Próxima (Perigo): Se a distância for menor que 40 polegadas, o LED Vermelho é aceso.

b) Distância Intermediária: Se a distância estiver entre 40 e 80 polegadas, o LED Azul é aceso.

c) Distância Longe (Segurança): Se a distância for maior que 80 polegadas, o LED Verde é aceso.

Monitoramento: Após a medição e a sinalização, os valores de distância são impressos no monitor serial. O processo se repete a cada 100 milissegundos.

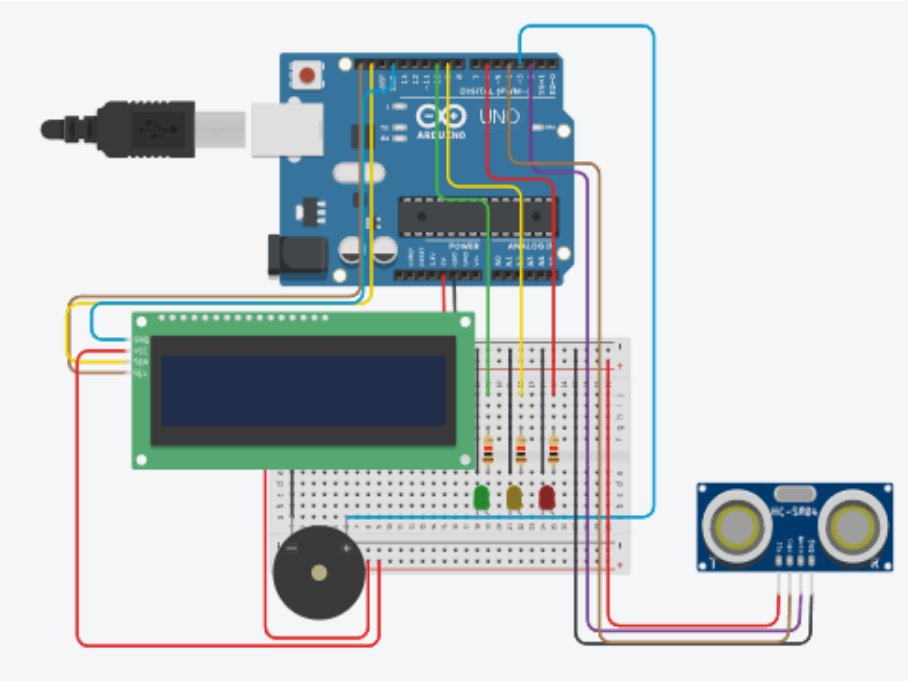
Figura 4.17 Construção do código infinito.

```
void loop() {
  long duration, inches, cm;
  pinMode(pingPin, OUTPUT);
  digitalWrite(pingPin, LOW);
  delayMicroseconds(2);
  digitalWrite(pingPin, HIGH);
  delayMicroseconds(5);
  digitalWrite(pingPin, LOW);
  pinMode(pingPin, INPUT);
  duration = pulseIn(pingPin, HIGH);

  // convert the time into a distance
  inches = microsecondsToInches(duration);
  cm = microsecondsToCentimeters(duration);
  if (inches < 40){
    digitalWrite (red,HIGH);
    digitalWrite (blue,LOW);
    digitalWrite (green,LOW);
  }
  if (inches > 80){
    digitalWrite (red,LOW);
    digitalWrite (blue,LOW);
    digitalWrite (green,HIGH);
  }
  if (inches > 40 &&  inches < 80){
    digitalWrite (red,LOW);
    digitalWrite (blue,HIGH);
    digitalWrite (green,LOW);
  }
  Serial.print(inches);
  Serial.print("in, ");
  Serial.print(cm);
```

Fonte: Elaborado pelo Próprio Autor

Na Figura 4.18 pode-se verificar a estrutura de um circuito de Sensor de Proximidade com Display, que foi montada e disponibilizada para desenvolvimento do código de programação em sala.

4.18 Figura Sensor de Proximidade com Display



Fonte: Autoria própria via software Tinkercad

O código começa incluindo as bibliotecas necessárias para a comunicação I2C e o controle do display LCD Figura 4.19. Em seguida, são declaradas as variáveis para os pinos dos componentes: o *buzzer* (pino 3), os LEDs (Vermelho no pino 6, Amarelo no 9 e Verde no 10), e uma variável para armazenar a distância medida (distancia).

Figura 4.19: Declaração de Variáveis

```
3
4  // Inicializa o LCD no endereço 0x27 com 16 colunas e 2 linhas
5  LiquidCrystal_I2C lcd(0x27, 16, 2);
6
7  int buzzer = 3;
8
9  int ledVermelho = 6;
10 int ledAmarelo = 9;
11 int ledVerde = 10;
```

Fonte: Elaborado pelo Próprio Autor

A função *setup*() inicializa a comunicação serial Figura 4.20. Também inicializa o LCD e liga sua luz de fundo. Os pinos do *buzzer* e de todos os três LEDs (ledVermelho, ledAmarelo, ledVerde) são configurados como saídas (OUTPUT), pois o Arduino enviará sinais para controlá-los.

Figura 4.20 Configuração de Output e Input circuito

```
13 int distancia;
14
15 long leitorDistancia(int triggerPin, int echoPin){
16   pinMode(triggerPin, OUTPUT);
17   digitalWrite(triggerPin, LOW);
18   delayMicroseconds(2);
19
20   digitalWrite(triggerPin, HIGH);
21   delayMicroseconds(10);
22   digitalWrite(triggerPin, LOW);
23   pinMode(echoPin, INPUT);
24
```

Fonte: Elaborado pelo Próprio Autor

A função *loop*() é o núcleo do programa. Ela calcula a distância chamando `leitorDistancia`(4,2) (usando o pino 4 como Trigger e o 2 como Echo) e converte o tempo para centímetros usando um fator de conversão46. A distância é exibida tanto no monitor serial quanto no display LCD47.O código implementa uma lógica de três zonas de alerta: Segurança (> 100 cm): Chama verde() (LED Verde ligado, *Buzzer* desligado)48.Atenção (70 cm a 100 cm): Chama amarelo() (LED Amarelo ligado, *Buzzer* desligado)49.Alerta (70 cm): Chama vermelho() (LED Vermelho ligado, *Buzzer* ligado)50. Assim feita na Figura 4.21 e Figura 4.22.

Figura 4.21 Construção do código infinito. Parte 1

```
28  void setup() {
29    Serial.begin(9600);
30
31    lcd.init();                  // Inicializa o LCD
32    lcd.backlight();             // Liga a luz de fundo
33
34    pinMode(buzzer, OUTPUT);
35    pinMode(ledVermelho, OUTPUT);
36    pinMode(ledAmarelo, OUTPUT);
37    pinMode(ledVerde, OUTPUT);
38  }
39
40  void loop() {
41    distancia = 0.01723 * leitorDistancia(4, 2);
42
43    Serial.print("Distancia: ");
44    Serial.print(distancia);
45    Serial.println("cm");
46
47    // Exibe a distância no LCD
48    lcd.clear();
49    lcd.setCursor(0, 0);
50    lcd.print("Distancia:");
51    lcd.setCursor(0, 1);
52    lcd.print(distancia);
53    lcd.print(" cm");
54
55    if(distancia > 100){
56      verde();
57    }
58    else if(distancia > 70 && distancia <= 100){
59      amarelo();
```

Fonte: Elaborado pelo Próprio Autor

Figura 4.22 Construção do código infinito Parte 2

```
60    }
61    else {
62      vermelho();
63    }
64
65    delay(200); // Pequeno atraso para evitar piscadas rápidas
66  }
67
68  void vermelho(){
69    digitalWrite(ledVerde, LOW);
70    digitalWrite(ledAmarelo, LOW);
71    digitalWrite(ledVermelho, HIGH);
72    digitalWrite(buzzer, HIGH);
73    delay(100);
74  }
75
76  void verde(){
77    digitalWrite(ledVerde, HIGH);
78    digitalWrite(ledAmarelo, LOW);
79    digitalWrite(ledVermelho, LOW);
80    digitalWrite(buzzer, LOW);
81    delay(100);
82  }
83
84  void amarelo(){
85    digitalWrite(ledVerde, LOW);
86    digitalWrite(ledAmarelo, HIGH);
87    digitalWrite(ledVermelho, LOW);
88    digitalWrite(buzzer, LOW);
89    delay(100);
90  }
```

Fonte: Elaborado pelo Próprio Autor

4.2 CONCLUSÃO

O estudo e desenvolvimento dos circuitos em Arduino, tema central deste capítulo, representaram uma experiência de grande valor. Por meio da execução prática desses circuitos, foi possível aplicar os conceitos e conhecimentos assimilados na disciplina. Foi enfrentado um desafio concreto, resultando no desenvolvimento de soluções eficazes e funcionais.

Mais do que um projeto isolado, a criação desses circuitos em Arduino marca um ponto significativo na trajetória de aprendizado. Eles permitiram consolidar e reunir, em uma única aplicação, todo o conhecimento técnico e as metodologias adquiridas.

5. CONEXÃO DE REDES DE COMPUTADORES

No quinto semestre do curso de Análise e Desenvolvimento de Sistemas (ADS), a disciplina de Redes de Computadores foi crucial para a formação de habilidades indispensáveis em Tecnologia da Informação. O conteúdo programático incluiu tópicos como: tipos de comunicação, classificações e topologias de redes, funcionamento de dispositivos como *Switch*, DHCP (*Dynamic Host Configuration Protocol*), Web Server, DNS (*Sistema de Nomes de Domínio*) e *Router*. Para reforçar o aprendizado, foi utilizado o *Cisco Packet Tracer*, um *software* de simulação de redes. Essa ferramenta possibilitou a criação e o teste de configurações, a experimentação com topologias e dispositivos variados, e a solução de desafios práticos. A prática com o *Packet Tracer* permitiu simular ambientes de rede e testar diferentes ajustes sem incorrer em riscos ou danos reais, facilitando significativamente a assimilação dos conceitos estudados.

5.1 TIPOS DE COMUNICAÇÃO

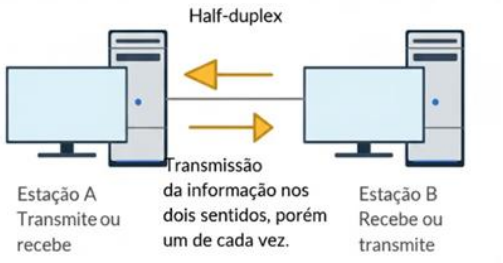
Simplex**:** Conforme Forouzan (2010), “é o método mais simples de transmissão, no qual a comunicação é estritamente unidirecional”. Um exemplo comum é o rádio, onde a estação apenas transmite e os receptores apenas escutam. Embora menos frequente em redes que exigem troca bidirecional, é apropriado para cenários em que a comunicação em sentido único é suficiente, como em transmissões de televisão.

Figura 5.1 Comunicação Simplex

Fonte: Elaborado pelo Próprio Autor

Half-Duplex**:** Conforme Forouzan (2010), “permite a comunicação bidirecional, porém não simultânea, ocorrendo a transmissão em apenas um sentido por vez”. O walkie-talkie é um exemplo típico, exigindo alternância entre envio e recepção. Em redes de computadores, o método é encontrado em Ethernets mais antigas ou em ambientes que utilizam controle de acesso para evitar colisões. É mais eficiente que o Simplex, pois possibilita troca de dados em ambos os sentidos.

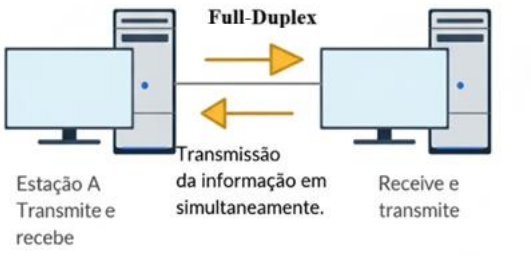
Figura 5.2 - Comunicação Half-duplex



Fonte: Elaborado pelo Próprio Autor

Full-Duplex: Conforme Forouzan (2010), “caracteriza-se por permitir a transmissão simultânea de dados nos dois sentidos”. Um exemplo cotidiano é o telefone, que possibilita a interação simultânea entre as partes. Nas arquiteturas de redes atuais, o Full-Duplex é amplamente utilizado, especialmente em tecnologias como Ethernet e Wi-Fi, por oferecer maior velocidade e eficiência. Essa modalidade maximiza a utilização da banda, reduz atrasos e melhora o desempenho da rede, sendo essencial em sistemas que exigem alta vazão e baixa latência, como streaming de vídeo e jogos em rede. A representação gráfica do Full-Duplex está ilustrada na Figura 5.3.

Figura 5.3 - Comunicação Full-duplex



Fonte: Elaborado pelo Próprio Autor

5.2 DISPOSITIVOS DE REDES

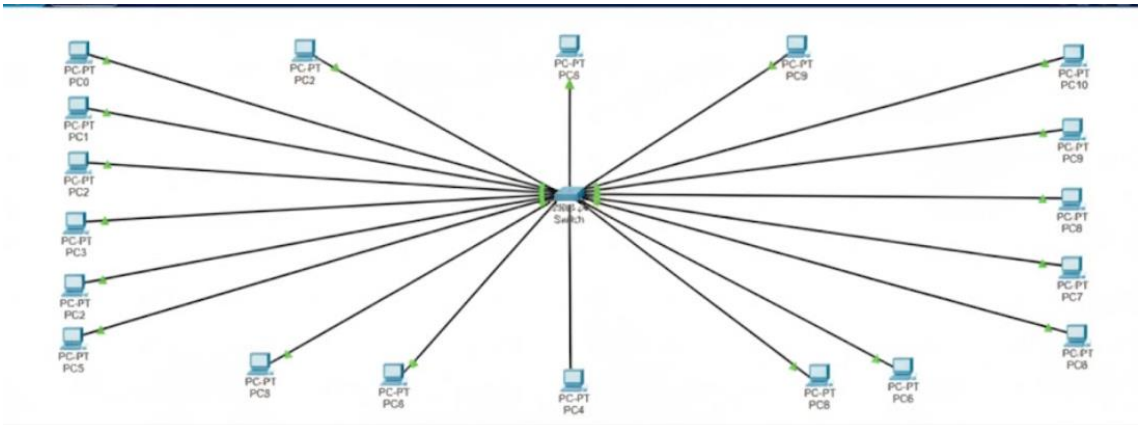
Desempenhando um papel crucial na comunicação entre sistemas em rede, os dispositivos de rede são essenciais para o funcionamento adequado de toda a infraestrutura de redes. Eles auxiliam no controle e direcionamento dos dados, garantindo a entrega correta das informações. Dispositivos específicos como *switches*, roteadores, *bridges* e pontos de acesso possuem funções vitais e diferenciadas para manter a rede em bom funcionamento. Conforme apresentado por Dantas (2002) ao tratar dos fundamentos de redes “incluindo modelos OSI/TCP-IP, protocolos, topologias e equipamentos” a compreensão do papel desses dispositivos é indispensável para o planejamento e a operação eficiente de qualquer ambiente de comunicação digital.

5.2.1 Switch

O *switch* é um dispositivo que une múltiplos aparelhos em uma LAN e encaminha pacotes de dados utilizando o endereço MAC de destino. Seu uso aumenta a eficiência e o desempenho da rede, possibilitando comunicação simultânea entre diversos dispositivos.

A Figura 5.4 ilustra um exemplo de *switch,* em uso que foi desenvolvido em aula, o objetivo era a comunicação entre os computadores utilizando o *switch.* O trabalho foi feito utilizando o prompt de comando que utiliza a linguagem em PHP, fazendo com que os computadores se comuniquem entre eles através do *switch*.

Figura 5.4 Exemplo prático de Switch



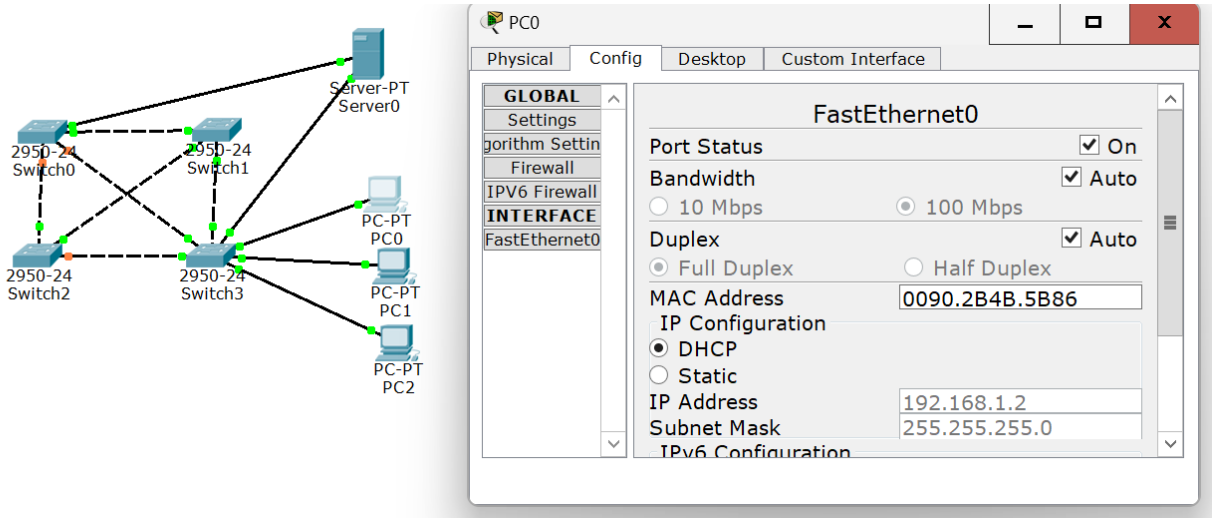
Fonte: Elaborado pelo Próprio Autor

5.2.2 DHCP (Dynamic Host Configuration Protocol)

O protocolo DHCP possibilita a atribuição dinâmica de endereços IP, garantindo que os dispositivos em uma rede sejam configurados de maneira autônoma. Isso resulta em uma gestão de rede mais descomplicada, pois elimina a tarefa de definir endereços IP de forma individual.

Um exemplo de uso prático do DHCP é ilustrado na Figura 5.5 onde foi desenvolvido em um trabalho com o objetivo de que os computadores tivessem acesso ao servidor e sua comunicação seja segura gerando de forma automática os endereços IP entre servidor, *switch* e computador fornecendo acesso ao servidor mesmo que um dos *switches* pare de funcionar.

Figura 5.5 Exemplo prático de DHCP (Dynamic Host Configuration Protocol)



Fonte: Elaborado pelo Próprio Autor

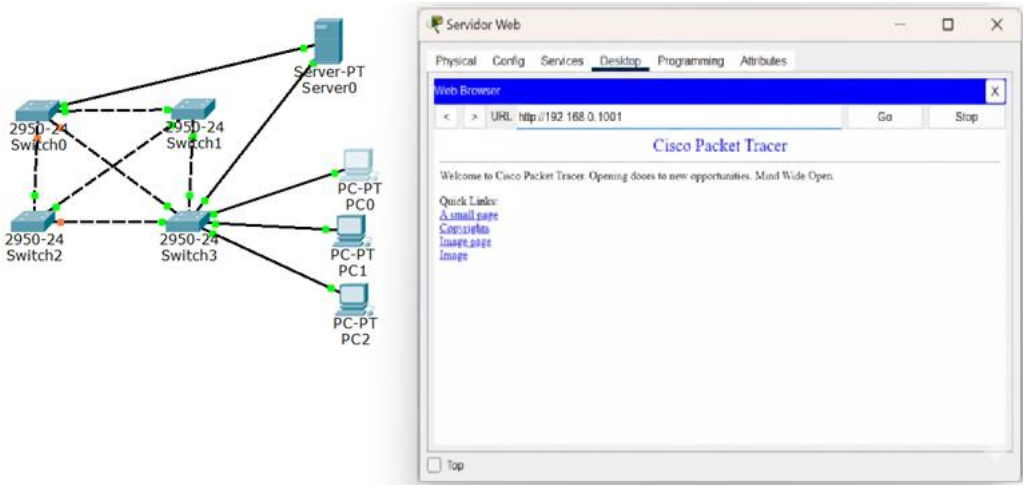
5.2.3 Web Server

O servidor web consiste no programa que tem como função principal entregar conteúdo de páginas web. Ele responde a comandos HTTP enviados pelos navegadores, disponibilizando os elementos solicitados, incluindo código HTML, folhas de estilo CSS e mídias visuais. Dominar a instalação e a gestão de servidores web é um conhecimento crucial para o profissional da área de TI (Tecnologia da Informação). Além disso, conforme discutido por Chee J. S. B. e Franklin C. Junior (2013), “a computação em nuvem ampliou significativamente as possibilidades de

implementação desses serviços, permitindo maior flexibilidade, escalabilidade e gerenciamento remoto”.

Um exemplo de um Web Server é apresentado na Figura 5.6, onde foi criada uma página web no servidor que só pode ser acessada pelos computadores graças à conexão entre o servidor, switches e os computadores.

Figura 5.6 Exemplo prático de Web Server



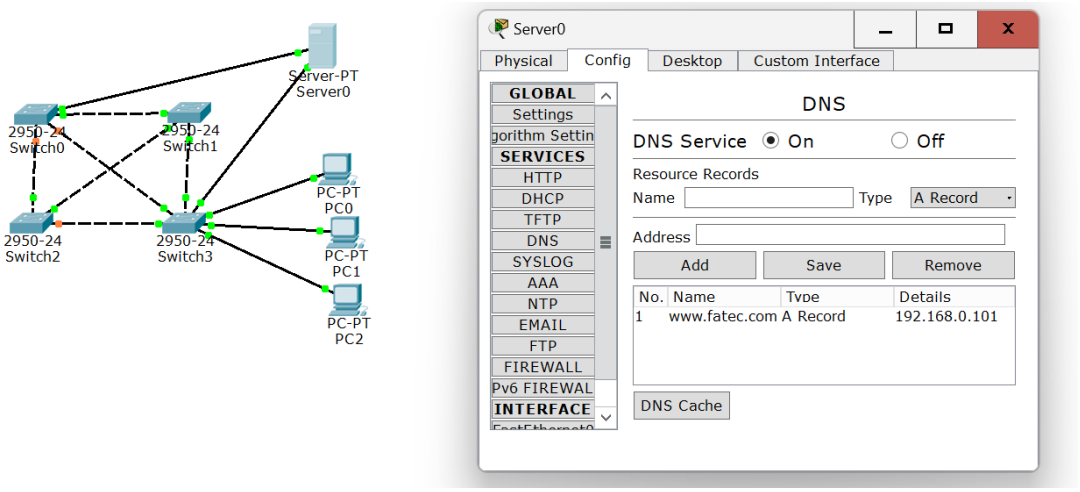
Fonte: Elaborado pelo Próprio Autor

5.2.4 Sistema de Nomes de Domínio (DNS)

O Sistema de Nomes de Domínio (DNS) é vital para a *web*, pois converte nomes de domínio (ex.: www.fatec.com) em endereços IP (ex.: 192.168.0.101). Quando uma URL é digitada, o DNS busca o IP do servidor correspondente. Por exemplo, ao tentar acessar um endereço, o DNS localiza o endereço IP correto, permitindo a conexão com o servidor.

A Figura 5.7 ela mostra a conversão do nome do domínio www.fatec.com em um endereço de IP 192.168.0.101 fornecendo a entrada a página pelos computadores conectados à rede.

Figura 5.7 Exemplo prático de 4 Sistema de Nomes de Domínio



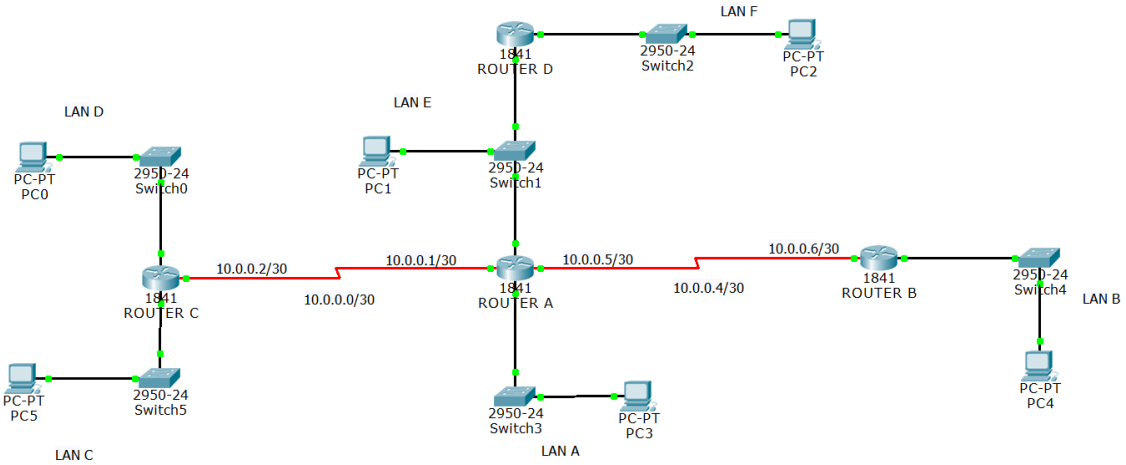
Fonte: Elaborado pelo Próprio Autor

5.2.5 Router

Este equipamento é responsável por direcionar o tráfego de dados entre múltiplas redes, baseando sua decisão no endereço IP de destino para identificar o trajeto ideal. Os roteadores são vitais para estabelecer a comunicação entre diferentes segmentos de rede, sejam eles internos ou remotos.

O *Router* ilustrado na Figura 5.8 tem como objetivo a conexão de uma Rade de computadores utilizando endereços IP distintos, além do uso de duas redes diferentes sendo elas 10.0.0.1/30 e 10.0.0.5/30 sendo a primeira a esquerda e a segunda à direita o objetivo foi estabelecer uma conexão segura e estável entre todos os computadores fazendo com que eles pudessem se comunicar entre si mesmo com redes e IPs diferentes.

Figura 5.8 Exemplo prático de Route



Fonte: Elaborado pelo Próprio Autor

5.3 CONCLUSÃO

O quinto semestre dedicado à disciplina de Redes de Computadores forneceu um alicerce robusto em conceitos e metodologias fundamentais. A integração entre a teoria e a prática permitiu desenvolver as competências indispensáveis para planejar, configurar e administrar redes de computadores de maneira eficiente.

6 GERENCIAMENTO DE USUÁRIOS

Neste capítulo, foi elaborada e desenvolvida uma aplicação web para gerenciamento de usuários, onde é realizado o CRUD (Create, Read, Update, Delete) para o usuário. A aplicação foi desenvolvida utilizando-se HTML, CSS, PHP e Banco de Dados MySQL. Conforme Machado (2014), “o projeto de banco de dados deve assegurar a organização lógica das informações, garantindo integridade, consistência e eficiência no acesso”. Assim, seguindo esse princípio, estruturou-se o banco de dados de modo que atendesse às necessidades da aplicação. Foi utilizada, também, uma biblioteca externa para estilização, chamada Bootstrap, aplicando técnicas e estudos necessários para que o sistema fosse adequado aos parâmetros estabelecidos além do uso de outras fermentas como Composer que é uma ferramenta de gerenciamento de dependências para PHP.

6.1 DESENVOLVIMENTO

Em alinhamento com os objetivos deste estudo, a disciplina Tópicos Especiais em Informática requisitou e supervisionou a criação da seção de gerenciamento de usuários para uma aplicação *web*, focada no cadastro de produtos e usuários. O projeto foi implementado utilizando as tecnologias HTML, CSS e PHP, com o gerenciamento de dados realizado através de Banco de Dados SQL. Para a estilização visual, foi empregada a biblioteca *Bootstrap*.

Nesta tela Figura 6.5, está disponível a lista de usuários com suas informações, sendo que cada um possui permissões específicas. O usuário com permissão Admin pode editar e excluir outros cadastros, enquanto o usuário Normal tem sua funcionalidade restrita apenas à edição.

6.2 COMPONENTES MVC DO MÓDULO

O módulo de gerenciamento de usuários segue o padrão MVC (Model-View-Controller). O fluxo de controle é orquestrado pelos seguintes componentes principais no *namespace* App\\:

Figura 6.1 Camadas/Componentes CRUD

Camada	Componente	Responsabilidade
Controller	App\Controllers\UsuarioController	Trata as requisições HTTP (Rotas), chama os métodos de Serviço e carrega as Views (Telas).
Model (Entidade)	App\Models\Entidades\Usuario	Representa a tabela dos usuários do banco de dados e suas permissões.
Model (DAO)	App\Models\DAO\UsuarioDAO	Acessa o banco de dados (CRUD).
Views	/App/Views/usuario/*	Renderizam as telas HTML (Listagem, Cadastro, Edição).

Fonte: Elaborado pelo Próprio Autor

O módulo de gerenciamento de usuários segue o padrão MVC(Model-View-Controller), utilizando os seguintes componentes em Camada, Componente, Responsabilidade *Controller*, `App\Controllers\UsuarioController`," Tratar as requisições HTTP (Rotas), chamar os métodos de Serviço e carregar as *Views*, a tabela dos usuários do banco de dados e suas permissões ao acessar o banco de dados (CRUD).

Figura 6.2 Código UsuarioController Parte 1

App > Controllers > UsuarioController.php

```php
<?php

namespace App\Controllers;
use App\Lib\Permissao;
use App\Lib\Sessao;
use App\Lib\Util;
use App\Models\DAO\UsuarioDAO;
use App\Models\Entidades\Usuario;
use App\Lib\Autenticacao;

class UsuarioController extends Controller
{
    public function listar()
    {
        Permissao::requerPermissao(['Admin']);
        Autenticacao::requerAutenticacao();

        $usuarioDAO = new UsuarioDAO();
        self::setViewParam('listaUsuarios', $usuarioDAO->listar());
        $this->render('/usuario/listar');
        Sessao::limpaMensagem();
    }

    public function cadastrar()
    {
        Permissao::requerPermissao(['Admin']);
        Autenticacao::requerAutenticacao();

        $this->render('/usuario/cadastrar');
        Sessao::limpaMensagem();
        Sessao::limpaErro();
    }

    public function salvar()
    {
        Permissao::requerPermissao(['Admin']);
        Autenticacao::requerAutenticacao();

        $dados = Util::sanitizar($_POST);
```

Fonte: Elaborado pelo Próprio Autor

Figura 6.3 Código UsuarioController Parte 2

App > Controllers > UsuarioController.php

```
12  {
58      public function editar($params)
59      {
60          Permissao::requerPermissao(['Admin']);
61          Autenticacao::requerAutenticacao();
62
63          $login = $params[0];
64          $usuarioDAO = new UsuarioDAO();
65          $usuario = $usuarioDAO->listar($login);
66
67          if (!$usuario) {
68              Sessao::gravaMensagem('<div class="alert alert-danger">Usuário não encontrado.</div>');
69              $this->redirect('/usuario/listar');
70          }
71
72          self::setViewParam('usuario', $usuario);
73          $this->render('/usuario/editar');
74          Sessao::limpaMensagem();
75          Sessao::limpaErro();
76      }
77
78      public function atualizar()
79      {
80          Permissao::requerPermissao(['Admin']);
81          Autenticacao::requerAutenticacao();
82
83          $dados = Util::sanitizar($_POST);
84          $usuario = new Usuario();
85          $usuario->setUsuario($dados);
86
87          $usuarioDAO = new UsuarioDAO();
88          $usuarioDAO->atualizar($usuario);
89
90          Sessao::gravaMensagem('<div class="alert alert-success">Usuário atualizado com sucesso.</div>');
91          $this->redirect('/usuario/listar');
92      }
```

Fonte: Elaborado pelo Próprio Autor

6.3 ESTABELECIMENTO DA CONEXÃO COM O BANCO DE DADOS

O processo para que o sistema possa realizar as operações CRUD inicia com a configuração e o estabelecimento da conexão com o banco de dados. A Figura 6.4 exibe o bloco de código responsável pela configuração e estabelecimento da conexão. O processo começa pela declaração de variáveis que contêm as informações essenciais: o *host* (link de acesso), usuário, senha e o nome do banco de dados (`crudproduto`). Em seguida, essas variáveis são integradas a um comando que cria a instância de conexão. A conexão é então executada, e o sistema é configurado para informar o usuário caso ocorra alguma falha.

Figura 6.4 - Conexão com o Banco de Dados

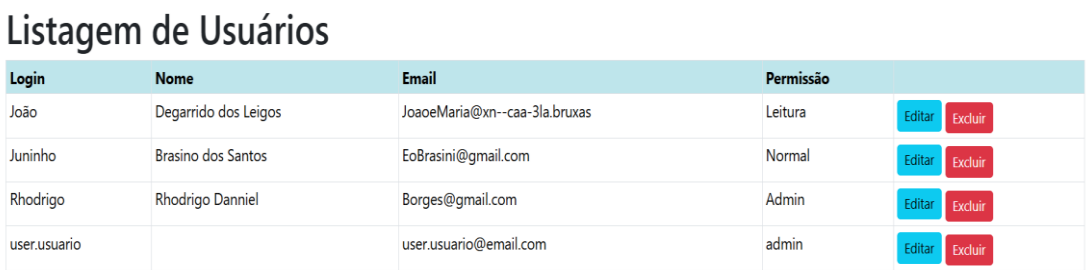
```php
<?php
  //Credenciais para conexão com o Banco
  $dbhost = 'localhost:3306';
  $dbuser = 'root';
  $dbpass = '';
  $dbname = 'crudproduto';

  //Abre conexão com o MySQL
  $conn = mysqli_connect($dbhost, $dbuser, $dbpass,$dbname);

  if(!$conn ){
    die('Falha ao conectar com o MySQL: ' . mysqli_connect_error());
  }
  //echo '<br>Conexão ao Banco realizada com Sucesso.';
  $sql = 'SELECT * FROM Usuario';
  $result = mysqli_query($conn, $sql); //A query seleciona as linhas da Tabela
```

Fonte: Elaborado pelo Próprio Autor

6.4 OPERAÇÃO READ (LISTAGEM DE USUÁRIOS)

A listagem de usuários é a interface de consulta (READ) do módulo. Nesta tela, está disponível a lista de usuários com suas informações, sendo que cada um possui permissões específicas que ditam as ações disponíveis (Edição e Exclusão).

Figura 6.5 Tela de Listagem de Usuário



Fonte: Elaborado pelo Próprio Autor

A interface aplica as seguintes regras de controle de acesso, baseadas no nível de permissão do usuário logado e do usuário listado:

a) Usuário Admin: Possui privilégios totais, podendo editar e excluir outros cadastros.

b) Usuário Normal: Tem sua funcionalidade restrita apenas à edição de registros (geralmente o próprio).

c) Permissão Leitura: Indica um nível de acesso mais restrito na aplicação, sem permissão para edição ou exclusão.

Figura 6.6 Código de Listagem de Usuário

App > Controllers > UsuarioController.php

```php
<?php

namespace App\Controllers;
use App\Lib\Permissao;
use App\Lib\Sessao;
use App\Lib\Util;
use App\Models\DAO\UsuarioDAO;
use App\Models\Entidades\Usuario;
use App\Lib\Autenticacao;

class UsuarioController extends Controller
{
    public function listar()
    {
        Permissao::requerPermissao(['Admin']);
        Autenticacao::requerAutenticacao();

        $usuarioDAO = new UsuarioDAO();
        self::setViewParam('listaUsuarios', $usuarioDAO->listar());
        $this->render('/usuario/listar');
        Sessao::limpaMensagem();
    }
```

Fonte: Elaborado pelo Próprio Autor

A lógica para buscar e exibir os usuários segue o padrão MVC, utilizando a classe `UsuarioController` para gerenciar a requisição. O código na Figura 6.3 exibe a estrutura principal do *Controller*, onde o método listar() é o responsável por:

a) Segurança: Requer autenticação e permissão de 'Admin' para acesso total à listagem.

b) Busca de Dados: Chama o `UsuarioDAO` (Data Access Object) para obter a lista de usuários do banco.

c) Visualização: Renderiza a `View` (/usuario/listar) e passa a lista de usuários como parâmetro.

Figura 6.7 - Código do Controller (UsuarioController: listar)

App > Controllers > UsuarioController.php

```php
<?php

namespace App\Controllers;
use App\Lib\Permissao;
use App\Lib\Sessao;
use App\Lib\Util;
use App\Models\DAO\UsuarioDAO;
use App\Models\Entidades\Usuario;
use App\Lib\Autenticacao;

class UsuarioController extends Controller
{
    public function listar()
    {
        Permissao::requerPermissao(['Admin']);
        Autenticacao::requerAutenticacao();

        $usuarioDAO = new UsuarioDAO();
        self::setViewParam('listaUsuarios', $usuarioDAO->listar());
        $this->render('/usuario/listar');
        Sessao::limpaMensagem();
    }
```

Fonte: Elaborado pelo Próprio Autor

O processo de listagem se complementa com a execução da *query* no banco e a construção da tabela na *View*, conforme detalhado nas figuras Figura 6.5. O trecho do código PHP que declara as variáveis de conexão (se repetidas) e executa a query SQL de busca.

Figura 6.8 - mySQL Listagem de Usuário

App > Views > usuario > listar.php

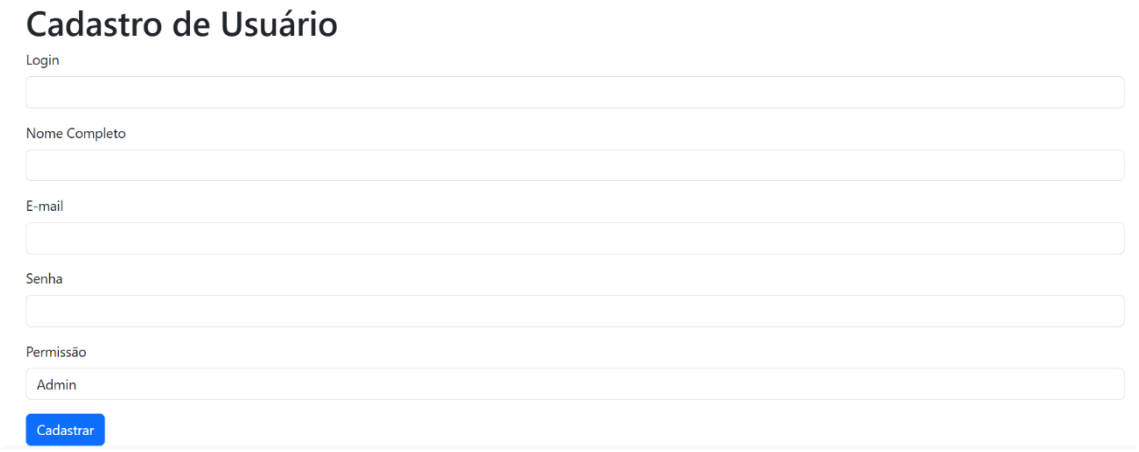
```php
<main role="main" class="flex-shrink-0">
  <div class="container">
    <h1 class="mt-5">Listagem de Usuários</h1>

    <?php
      echo $Sessao::retornaMensagem();
      $Sessao::limpaMensagem();

      if (count($viewVar['listaUsuarios']) > 0) {
        echo '<div class="table-responsive">';
        echo '  <table class="table table-bordered table-hover table-sm">';
        echo '    <thead>';
        echo '      <tr>';
        echo '        <th class="table-info">Login</th>';
        echo '        <th class="table-info">Nome</th>';
        echo '        <th class="table-info">Email</th>';
        echo '        <th class="table-info">Permissão</th>';
        echo '        <th class="table-info"></th>';
        echo '      </tr>';
        echo '    </thead>';
        echo '    <tbody>';

        foreach ($viewVar['listaUsuarios'] as $usuario) {
          echo '<tr>';
          echo '  <td>' . $usuario->getLogin() . '</td>';
          echo '  <td>' . $usuario->getNome() . '</td>';
          echo '  <td>' . $usuario->getEmail() . '</td>';
          echo '  <td>' . $usuario->getPermissao() . '</td>';
          echo '  <td>';
          echo '    <a href="http://' . APP_HOST . '/usuario/editar/' . $usuario->getLogin() . '" class="btn btn-info btn-sm">Editar</a> ';
          echo '    <a href="http://' . APP_HOST . '/usuario/excluirConfirma/' . $usuario->getLogin() . '" class="btn btn-danger btn-sm mt-1">Excluir</a>';
          echo '  </td>';
          echo '</tr>';
        }

        echo '    </tbody>';
        echo '  </table>';
        echo '</div>';
      } else {
        echo '<div class="alert alert-info">Nenhum usuário encontrado.</div>';
      }
    ?>
  </div>
</main>
```

Fonte: Elaborado pelo Próprio Autor

6.5 OPERAÇÃO CREATE (CADASTRO DE NOVO USUÁRIO)

A funcionalidade de Cadastro de Usuário permite inserir novos registros na base de dados, exigindo permissão de administrador (Admin).

Figura 6.9 - Tela do Formulário de Cadastro



Fonte: Elaborado pelo Próprio Autor

O trecho de código na Figura 6.10 é responsável por capturar os dados submetidos pelo formulário e armazená-los em variáveis temporárias. Em seguida, é realizada uma validação essencial: caso o valor da permissão fornecida seja diferente de qualquer uma das três opções válidas (ADMIN, NORMAL, LEITURA), será exibida uma mensagem de "Permissão inválida", interrompendo a execução.

Figura 6.10 - Cadastro: Coleta de Dados e Verificação de Permissão

```php
// Recebendo os dados do formulário
$login = $_POST['login'];
$nome = $_POST['nome'];
$senha = $_POST['senha'];
$email = $_POST['email'];
$permissao = $_POST['permissao'];

// Verifica se a permissão é válida
$permissoes_validas = array('ADMIN', 'NORMAL', 'LEITURA');
if (!in_array($permissao, $permissoes_validas)) {
    echo "Permissão inválida!";
    exit(); // Termina a execução do script se a permissão for inválida
}
```

Fonte: Elaborado pelo Próprio Autor

Neste bloco de código Figura 6.11, é criado o comando SQL de inserção e, posteriormente, ele é executado, emitindo a mensagem de sucesso ou fracasso.

Figura 6.11 - Cadastro: Código de Inserção no Banco de Dados

```php
// Inserindo os dados na tabela do banco de dados
$sql = "INSERT INTO usuario (login, nome, senha, email, permissao) VALUES ('$login', '$nome', '$senha', '$email', '$permissao')";

if ($conexao->query($sql) === TRUE) {
    echo "Usuário cadastrado com sucesso!";
} else {
    echo "Erro ao cadastrar usuário: " . $conexao->error;
}

// Fechando a conexão com o banco de dados
$conexao->close();
```

Fonte: Elaborado pelo Próprio Autor

6.6 OPERAÇÃO UPDATE (EDIÇÃO DE USUÁRIO)

O código apresentado na Figura 6.12 realiza a captação do *login* correspondente ao usuário que será alterado, geralmente via parâmetro GET na URL. Em seguida, é construída e executada uma consulta SQL no banco de dados. A resposta da consulta é tratada: o registro do usuário é salvo em uma variável se for encontrado, ou é emitida uma notificação de "usuário não encontrado".

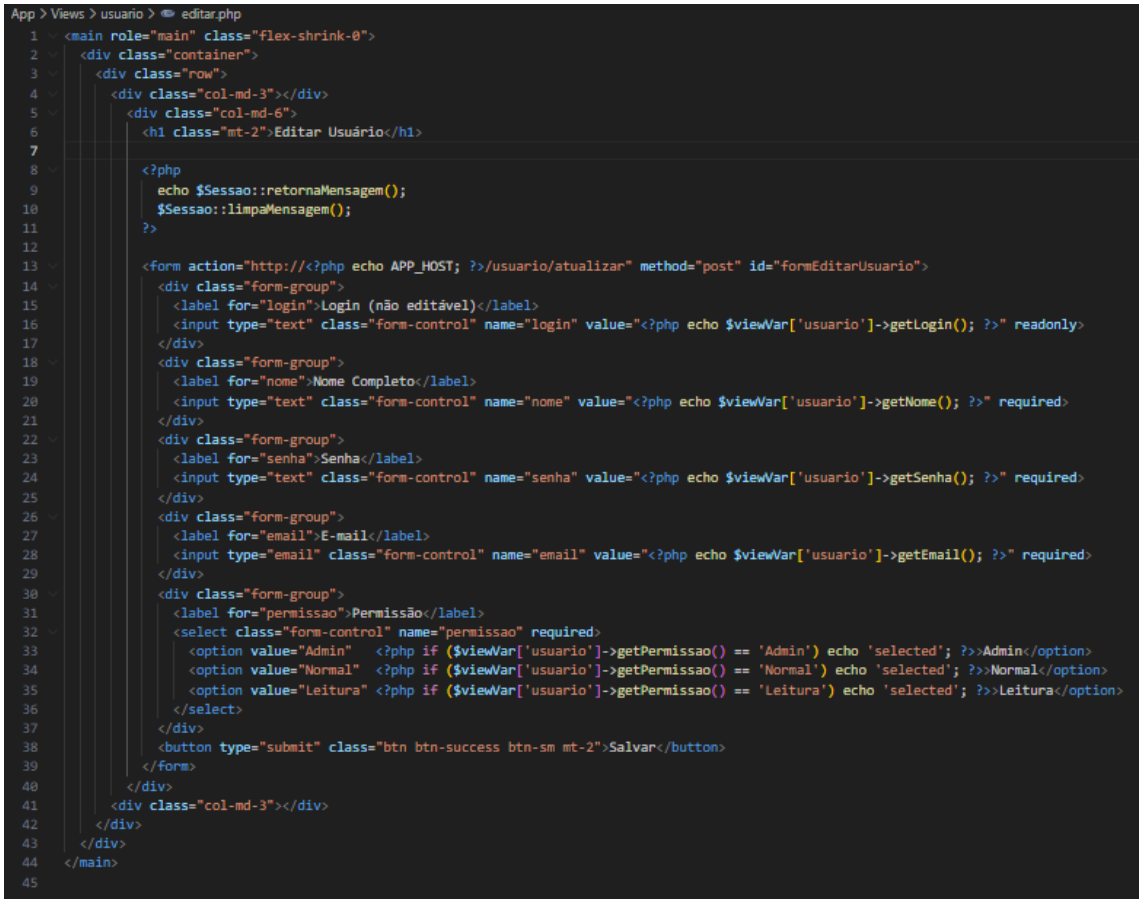
Figura 6.12 - Edição: Código de Busca do Usuário

```php
// Obtém o ID do usuário da URL
$login = $_GET['login'];

// Consulta SQL para obter os dados do usuário com o Login fornecido
$sql = "SELECT * FROM usuario WHERE login='$login'";
$resultado = $conexao->query($sql);

if ($resultado->num_rows > 0) {
    $usuarios = $resultado->fetch_assoc();

} else {
    echo "Usuário não encontrado";
}
```

Fonte: Elaborado pelo Próprio Autor

A finalidade do código na Figura 6.13 é coletar as informações que foram modificadas no formulário, construir o comando SQL de atualização (UPDATE) e executá-lo no banco de dados. O sistema notifica o usuário sobre o sucesso ou o fracasso da operação e, em seguida, redireciona a aplicação para a página que lista os usuários.

Figura 6.13 - Edição: Verificação de Formulário e Código de Atualização



Fonte: Elaborado pelo Próprio Autor

6.7 OPERAÇÃO DELETE (EXCLUSÃO DE USUÁRIO)

Na Figura 6.14, o código é responsável por efetuar a remoção de um usuário específico. É utilizado o *login* previamente armazenado para criar a instrução SQL de exclusão (DELETE) e, em seguida, ela é executada. A conclusão da operação é comunicada ao usuário por meio de uma notificação de êxito ou insucesso.

Figura 6.14 - Código de Exclusão de Usuário

App > Views > usuario > excluirConfirma.php

```php
<main role="main" class="flex-shrink-0">
  <div class="container">
    <div class="row">
      <div class="col-md-3"></div>
      <div class="col-md-6">
        <h1 class="mt-2">Exclusão de Usuário</h1>

        <form action="http://<?php echo APP_HOST; ?>/usuario/excluir" method="post" id="formExcluirUsuario">
          <input type="hidden" name="login" value="<?php echo $viewVar['usuario']->getLogin(); ?>">

          <div class="card text-white bg-danger mb-3" style="max-width: 22rem;">
            <div class="card-header">Confirmação da Exclusão</div>
            <div class="card-body">
              <h5 class="card-title">Excluir Usuário?</h5>
              Tem certeza que deseja excluir o usuário:
              <strong><?php echo $viewVar['usuario']->getNome(); ?></strong>
              (Login: <em><?php echo $viewVar['usuario']->getLogin(); ?></em>)?

              <div class="mt-3">
                <button type="submit" class="btn btn-primary btn-sm">Confirmar</button>
                <a href="http://<?php echo APP_HOST; ?>/usuario/listar" class="btn btn-secondary btn-sm">Cancelar</a>
              </div>
            </div>
          </div>
        </form>
      </div>
      <div class="col-md-3"></div>
    </div>
  </div>
</main>
```

Fonte: Elaborado pelo Próprio Autor

6.8 CONCLUSÃO

O desenvolvimento de uma aplicação *web* voltada para o gerenciamento de usuários, tema central deste capítulo, constituiu uma experiência extremamente enriquecedora. A realização deste projeto permitiu a aplicação prática e efetiva dos conhecimentos adquiridos na disciplina, possibilitando o enfrentamento de um desafio real e a criação de soluções eficientes.

Esta aplicação *web*, que vai além de um simples sistema de registro, representa um marco significativo na trajetória de aprendizado. Ela serviu para consolidar e unificar, em uma única plataforma, todas as técnicas e o conhecimento técnico acumulado.

7 MANUAL DO PORTIFÓLIO

Este manual tem como objetivo orientar os usuários na navegação e visualização das principais informações e competências técnicas demonstradas neste portfólio digital.

7.1 PORTFÓLIO DIGITAL

Diferente de um site com múltiplas páginas, este projeto é uma Página Única (Single Page Application - SPA), focada na apresentação concisa das suas habilidades.

Figura 7 – Portfólio Digital

Fonte: Elaborado pelo Próprio Autor

7.2 ESTRUTURA TÉCNICA E DESIGN

O portfólio é estruturado em um único arquivo HTML (index.html), o que simplifica o carregamento e garante alta performance. Tecnologias Core: HTML5 para a estrutura e Tailwind CSS para o design e a responsividade. Estilização: O design moderno, com cantos arredondados e cores claras de fundo `(#f1f5f9)`, é inteiramente definido pelas classes utilitárias do Tailwind CSS, carregado via CDN.

Responsividade: O layout é totalmente responsivo, garantindo que as informações se adaptem automaticamente a qualquer dispositivo, desde smartphones até monitores de desktop, reconfigurando os elementos grid conforme o tamanho da tela. Ícones:

Os ícones das tecnologias (como HTML5, JavaScript e o ícone de servidor para XAMPP) são importados da biblioteca Boxicons (via CDN), garantindo clareza visual e leveza.

Figura 7.1 – Estrutura Inicial / Tailwind CSS

```
<!DOCTYPE html>
<html lang="pt-BR">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Rhodrigo Danniel - Portfólio ADS Fatec</title>
    <!-- Tailwind CSS CDN -->
    <script src="https://cdn.tailwindcss.com"></script>
    <!-- Boxicons CDN for Icons -->
    <link href="https://unpkg.com/boxicons@2.1.4/css/boxicons.min.css" rel="stylesheet">
    <!-- Google Fonts - Inter -->
    <link href="https://fonts.googleapis.com/css2?family=Inter:wght@100..900&display=swap" rel="stylesheet">

    <!-- Configuração do Tailwind para cores personalizadas e fonte -->
    <script>
        tailwind.config = {
            theme: {
                extend: {
                    colors: {
                        'dark-bg': '#FFFCFC', // Cor de fundo principal
                        'primary': '#28B84B', // Cor principal (verde vibrante)
                        'secondary': '#1e293b', // Cor de fundo secundária para boxes e modais
                        'text-color': '#f1f5f9', // Cor do texto principal
                        'css-blue': '#28bcd6', // Azul do CSS
                        'html-orange': '#f07713',  // Azul do C++
                        'csharp-purple': '#9a48be', // Roxo do C#
                        'php-purple': '#777BB4', // Roxo do PHP
```

Fonte: Elaborado pelo Próprio Autor

7.3 SEÇÃO HABILIDADES

Esta é a seção principal do portfólio, apresentando as competências técnicas do desenvolvedor. Estrutura: As habilidades são organizadas em um layout de grid (4 colunas em telas grandes), exibindo um total de 8 tecnologias: HTML5, CSS, C++, C#, PHP, JavaScript, SQL e XAMPP. Conteúdo: Cada card de habilidade apresenta um ícone específico, o nome da tecnologia e uma descrição objetiva da sua aplicação no desenvolvimento. Interatividade: Ao passar o mouse sobre cada card, um efeito de destaque é ativado, com uma borda na cor verde neon (`#02E033`) e uma leve elevação, indicando a interatividade do elemento.

Figura 7.2 – Layout Minhas Habilidades

Fonte: Elaborado pelo Próprio Autor

Figura 7.3 – CSS/ Minhas Habilidades

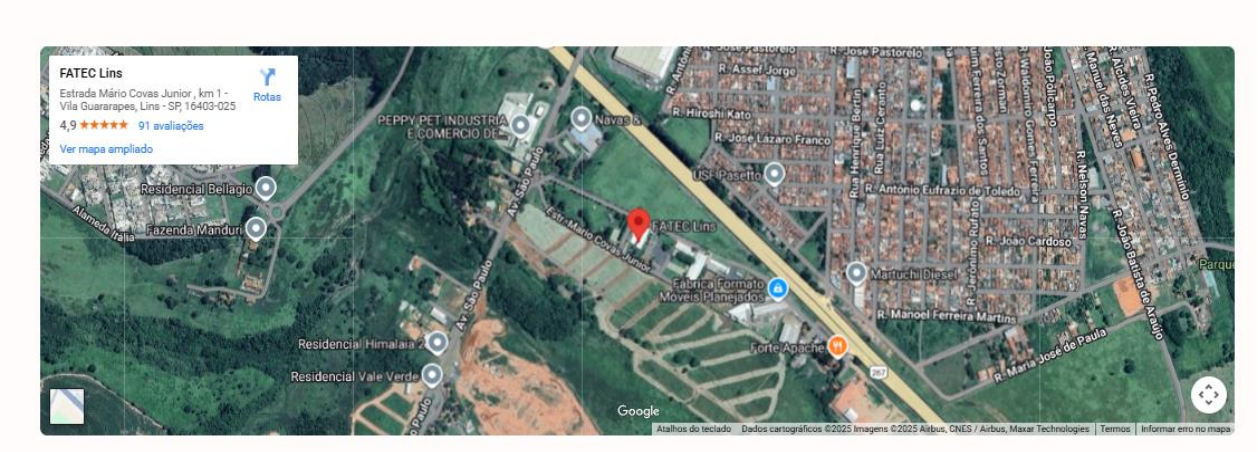
```css
.Habilidades-box {
    background: #1e1e3b;
    padding: 2rem;
    border-radius: 0.75rem;
    /* text-align: center; (Removido para alinhar o conteúdo à esquerda, como na imagem 1) */
    text-align: left;
    /* max-width: 300px; (Removido para permitir que o grid controle a largura) */
    transition: transform 0.3s ease;
    border: 2px solid transparent;
    color: white;
    /* Garante que o texto dentro da caixa seja branco */

    /* PROPRIEDADES NECESSÁRIAS PARA O LAYOUT */
    display: flex;
    flex-direction: column;
    gap: 0.5rem;
}
```

Fonte: Elaborado pelo Próprio Autor

7.4 SEÇÃO LOCALIZAÇÃO (MAPA)

Localizada abaixo da seção de habilidades, esta área tem a função de fornecer um ponto de referência físico ou acadêmico. Conteúdo: É um mapa interativo incorporado diretamente através de um elemento <iframe> do Google Maps. Visualização: O mapa está configurado para ocupar a largura total da tela, garantindo que ele se "una" de forma fluida ao rodapé, minimizando o espaçamento vertical entre as seções. O mapa está centrado na localização da FATEC Lins.

Figura 7.4 – Mapa Fatec Lins

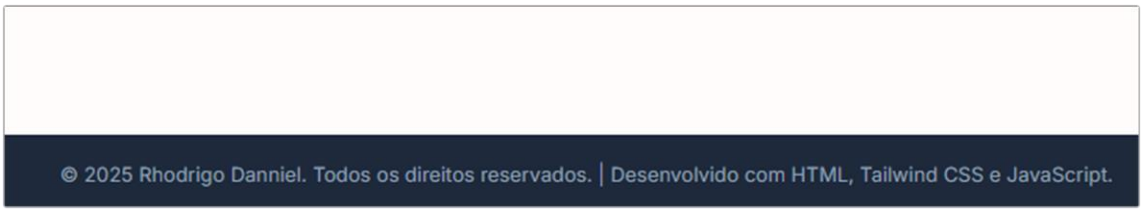


Fonte: Elaborado pelo Próprio Autor

7.5 RODAPÉ (*FOOTER*)

O rodapé é a última seção do portfólio e está fixado na parte inferior da tela. Informações: Contém a notificação de direitos autorais (© 2025 Rhodrigo Danniel. Todos os direitos reservados.) e a menção das tecnologias de desenvolvimento utilizadas no projeto (HTML, Tailwind CSS e JavaScript). Função: Garante o encerramento visual da página, mantendo o aspecto profissional.

Figura 7.5 – RODAPÉ



Fonte: Elaborado pelo Próprio Autor

8 CONCLUSÃO GERAL

O presente Portfólio de Projetos Acadêmicos encerra a trajetória no curso de Análise e Desenvolvimento de Sistemas (ADS), validando a formação de um profissional apto a atuar em todas as fases do ciclo de vida do desenvolvimento de software. Este trabalho demonstrou a proficiência na aplicação integral de conceitos teóricos em soluções práticas e funcionais, abrangendo os pilares essenciais da área de Tecnologia da Informação.

A escolha dessas matérias, Linguagem de Programação, Estrutura de Dados, Programação Orientada a Objetos Avançada, Redes de Computadores e Tópicos Especiais em Informática foi integralmente justificada por sua criticidade na formação técnica. A metodologia didática, pautada na combinação de fundamentos teóricos robustos com a prática intensa em projetos (como o Sistema de Recebimento, a Gestão de Livraria, os Circuitos em Arduino e a Aplicação Web CRUD), permitiu a consolidação do aprendizado de forma progressiva e aplicada.Os resultados obtidos em cada etapa destacam a capacidade de: Dominar a lógica de programação e desenvolver sistemas eficientes em linguagens de baixo e alto nível (C e C++). Modelar o raciocínio em estruturas de dados eficientes e garantir a persistência segura das informações. Construir soluções modulares e robustas, integrando hardware e software, como demonstrado nos Circuitos em Arduino. Compreender a infraestrutura de comunicação em rede, fundamental para que os sistemas operem de forma eficiente e segura. Atuar no Desenvolvimento Web Full-Stack, integrando front-end (HTML, CSS, Bootstrap), back-end (PHP) e banco de dados (MySQL) para entregar soluções seguras e prontas para o ambiente de produção.

Em suma, este portfólio não apenas documenta, mas também demonstra a proficiência técnica e a aptidão profissional para enfrentar o ciclo completo de desenvolvimento de software, entregando soluções funcionais, seguras e preparadas para operar em ambientes de rede modernos. A estruturação final do conteúdo em um Portfólio Digital (HTML, CSS e JavaScript) reforça a competência em todas as camadas da arquitetura de software.

REFERÊNCIAS

CHEE J. S. B.; JUNIOR FRANKLIN C.; **Computação em Nuvem** - Cloud Computing Tecnologias e Estratégias. São Paulo: M. Books. 2013.

RITCHIE, D. M. (1993). **The Development of the C Language**. Publicado nos Proceedings of the Second History of Programming Languages Conference (HOPL-II).

DANTAS, MARIO. **Tecnologias de Redes de Comunicação e Computadores**. 1. ed. Rio de Janeiro: Axcel Books: 2002. <link> https://www.feesc.org.br/site/?pg=trcc

ELMASRI, RAMEZ; NAVATHE, SHAMCANT B. **Sistemas de banco de dados**. 6 ed. São Paulo: Pearson Addison Wesley, 2011.

FREEMAM, ERIC . **Use a Cabeça! Programação JavaScript**. Alta Books: 24 maio 2016

FORBELLONE, A.L.V.;EBERSPACHER, H.F. **Lógica de Programação**. 3 ed. São Paulo: Pearson Prentice Hall, 2005.

FOROUZAN, BEHROUZ A. **Comunicação de Dados e Redes de Computadores**. 4 ed. Porto Alegre: AMGH, 2010.

MACHADO, FELIPE NERY RODRIGUES. **Banco de dados: Projetos e Implementação**. 3 ed. São Paulo: Érica, 2014.

DAVIS. STEPHEN R. **C++ Para Leigos**. Rio de Janeiro. Alta Books: 2016

VERAS, MANOEL. **Virtualização**: Tecnologia Central do Datacenter. Rio de Janeiro: Brasport, 2016.