Algoritmia e Programação
Trabalho Prático Nº2

Sistema de Gestão de Clínicas

Daniel Ventura Brito, Nº 32749

Docentes Orientadores:

David Luís Malhão Verde

João Filipe Correia Pereira

2024/2025

Cofinanciado por:

NORTE 2020 PROGRAMA OPERACIONAL REGIONAL DO NORTE

PORTUGAL 2020

UNIÃO EUROPEIA Fundo Social Europeu

Índice

1 Introdução

Foi desenvolvido, em Linguagem C, um sistema de gestão de clínicas, com o objetivo de controlar e gerir os dados dos seus clientes, médicos e consultas da clínica. O trabalho prático foi realizado no âmbito da unidade curricular de Algoritmia e Programação, durante o ano letivo 2024/2025, representa a evolução do primeiro trabalho prático apresentado, com o objetivo de consiliar os conhecimentos adquiridos e aprimorar habilidades, foram implementadas novas funcionalidades e adotando técnica de otimização de código, resultando em um projeto mais completo e de fácil leitura.

No presente relatório, iremos abordar ao detalhe os conceitos utilizados no projeto como vetores estáticos, as funções específicas, a manipulação de ficheiros e estruturas, apontadores e programação modular.

1.1 Obejtivo do Projeto

O objetivo do projeto é o desenvolvimento de uma aplicação para gestão de clientes, médicos e consultas de uma clínica, proporcionando uma automatização e simplicidade no processo de gerenciamento das operações diárias de uma clínica. A aplicação foi implementada para gerenciar as informações dos clientes, médicos e consultas, permitindo a fácil inserção, alteração e procura desses dados. Além disso, oferece opções de consulta de dados, como a disponibilidade de médicos, o histórico de consultas de clientes, o que demonstram uma utilização eficiente.

A interação com a aplicação é simples e intuitiva, com uma interface de linha de comandos (CLI), onde o utilizador pode interagir com a aplicação por meio de algarismos correspondendo a funções específicas.

2 Estrutura do Projeto

O Sistema de Gestão de Clínicas foi desenvolvido utilizando a técnica de programação modular, dividindo o projeto por módulos, visando a organização e facilidade na manutenção do código. O projeto é constituído por várias secções, como os clientes, médicos, consultas e auxiliares, cada uma implementada de forma independente.

2.1 Organização dos Diretórios

A organização dos diretórios do projeto foi planeada para facilitar a compreensão do desenvolvimento e otimizar o trabalho, a estrutura adotada segue um padrão comum na comunidade de desenvolvimento, visando a sepração de ficheiros com diferentes funções.

Tabela 1: Estrutura do Projeto

Ficheiro	Descrição
src/	Código-fonte do projeto, incluí os módulos.
inc/	Ficheiros header, constitui a declaração de funções.
data/	Fcheiros de dados, armazena dados relevantes do projeto.
tests/	Ficheiros de teste, verificação de funções específicas.
docs/	Documentação, incluí o relatório do trabalho prático.

2.2 Módulos

O projeto foi organizado por diversos módulos, cada um com uma resposabilidade bem definida. Foram criados individualmente módulos para os clientes, médicos e consultas, onde cada módulo é resposável pela manipulação das informações dos clientes, médicos e consultas, permitindo a inserção, alteração e consulta de dados dos mesmos.

Além disso, secções para funções auxiliares e para os menus interativos foram implementadas, de modo a auxiliar o restante projeto.

Tabela 2: Módulos do Projeto

Ficheiro	Descrição
main.c	Declaração dos vetores das estruturas e as variáveis.
cliente.c	Implementação das funções relacionadas com os clientes.
cliente.h	Protótipos das funções relacionadas com os clientes.
medico.c	Implementação das funções relacionadas com os médicos.
medico.h	Protótipos das funções relacionadas com os médicos.
consulta.c	Implementação das funções relacionadas com as consultas.
consulta.h	Protótipos das funções relacionadas com as consultas.
menu.c	Implementação das funções dos menus.
menu.h	Protótipos das funções dos menus.
auxiliares.c	Implementação das funções auxiliares.
auxiliares.h	Protótipos das funções auxiliares.
structs.h	Implementação das estruturas e constantes.

2.3 Diretivas de Compilação

Concluído o capítulo da estruturação do projeto, a utilização de diretivas de compilação como ifndef, define e endif é essencial para garantir eficiência e segurança do código, principalmente na utilização de vários ficheiros headers. As diretivas são instruções que o compilador interpreta durante a fase de pré-processamento do código, permitindo o controlo do comportamento da compilação e evitando problemas como inclusão múltipla do mesmo ficheiro.

A diretiva ifndef faz a verificação de uma macro, se ainda não foi definida, a diretiva define faz a declaração da macro e o código entre ifndef e o endif é incluído pelo compilador. Sendo assim é evitado que os mesmos ficheiros headers sejam processados novamente. A aplicação e a compreensão das diretivas de compilação foram fundamentais para organização e segurança do projeto.

3 Estruturas de Dados

Neste capítulo, são apresentados as estruturas de dados implementadas no desenvolvimento do sistema de gestão de clínicas. Essas estruturas foram a principal meio na organização e manipulação eficiente das informações relacionadas com os clientes, médicos e consultas, além de incluir informações auxiliares como a morada e data para uma maior modularização e leitura no código.

Nas próximas secções, detalhamos cada uma das estruturas utilizadas, destacando seus elementos, tipos de dados e as relações entre elas.

3.1 Estrutura dos Clientes

A estrutura nomeada de ‘ST_CLIENTE’, é utilizada para armazenar os dados dos clientes. É composto por número de identificação único, atribuindo automaticamente com base na ordem de inserção, o nome do cliente e o e-mail, ambos limitados a 100 caracteres, a atribuição de uma data de nascimento do tipo de dado de uma outra estrutura designada de ‘ST_DATA’, a morada do cliente também do tipo de dado de uma outra estrutura ‘ST_MORADA’, ainda a declaração de tipos de dados inteiros como número de identificação fiscal (NIF), o número de utente (SNS), e ainda o estado de atividade do cliente atribuído como um tipo booleano, permitindo apenas dois valores (TRUE ou FALSE).

```
typedef struct {
unsigned int ID;
char nome[STRING_MAX];
char email[STRING_MAX];
ST_MORADA morada;
ST_DATA data_nascimento;
unsigned int long NIF;
unsigned int long SNS;
bool estado;
}ST_CLIENTE;
```

Tabela 3: Representação da estrutura dos clientes

Campo	Tipo	Descrição
ID	Inteiro Positivo	Identificação única do cliente.
nome	String	Nome do cliente.
morada	ST_MORADA	Conjunto de dados relevantes a morada.
data_de_nascimento	ST_DATA	Conjunto de valores inteiros relacionados com datas.
NIF	Long Inteiro Positivo	Identificação fiscal do cliente.
SNS	Long Inteiro Positivo	Número de utente do cliente.
estado	Booleano	Booleano representado o estado de atividade.

3.2 Estrutura dos Médicos

De igual forma, foi implementada uma estrutura designada 'ST_MEDICO', concebida para armazenar informações sobre os profissionais de saúde da clínica. A estrutura é composta por os seguintes campos, um número de identificação único para cada profissional, o nome do médico, uma especialidade, e ainda o indicador do estado de atividade do médico, atribuído com valores booleanos.

```
typedef struct {
unsigned int ID;
char nome[STRING_MAX];
char especialidade[STRING_MAX];
bool estado;
}ST_MEDICO;
```

Tabela 4: Representação da estrutura dos médicos

Campo	Tipo	Descrição
ID	Inteiro Positivo	Identificação única do médico.
nome	String	Nome do médico.
especialidade	String	Especifica a área do médico.
estado	Booleano	Referentes ao estado de atividade.

3.3 Estrutura das Consultas

A estrutura de dados para as consultas, responsável pelo agendamento e interações entre os médicos e os clientes. Cada consulta é composta por um número de identificação único para facilitar o rastreio, o horário que pertence à estrutura 'ST_DATA', utilizando para verificação de sobreposição de horas, um apontador para a estrutura dos clientes, um apontador para a estrutura dos médicos, permitindo referenciar diretamente um cliente e médico à consulta, e ainda o indicador do estado da consulta, utilizando um tipo enum para melhor legibilidade e restrição nas opções possíveis.

```
typedef struct {
unsigned int ID;
ST_CLIENTE *cliente;
ST_MEDICO *medico;
ST_DATA data_inicial;
ST_DATA data_final;
ESTADO estado;
}ST_CONSULTA;
```

Tabela 5: Representação da estrutura das consultas

Campo	Tipo	Descrição
ID	Inteiro Positivo	Identificação única da consulta.
*cliente	Apontador	Apontador para uma variável de ST_CLIENTE.
*medico	Apontador	Apontador para uma variável de ST_MEDICO.
data_inicial	ST_DATA	Data e hora do início da consulta.
data_final	ST_DATA	Data e hora do final da consulta.
estado	ENUM	Enumeração para representar o estado.

3.4 Estruturas Auxiliares

Adicionalmente, foram implementadas outras estruturas e enumeradores de modo a facilitar e organizar o conjunto de dados, garantindo uma eficiência na legibilidade no código.

3.4.1 Enumerador para o Estado

Foi também implementado um enumerador, utilizado para representar os possíveis estados de uma consulta, associando o estado a valores inteiros, tornando o código mais claro.

```
typedef enum {
Cancelado = 0,
Agendado = 1,
Realizado = 2
}ESTADO;
```

3.4.2 Estrutura para a Morada

A criação de uma estrutura auxiliar para uma representação organizada de informações de morada associadas aos clientes. A estrutura nomeada de 'ST_MORADA' é composta por o nome da rua onde o cliente reside, o número de código postal e ainda o nome da cidade.

```
typedef struct {
char rua[STRING_MAX];
unsigned long int codigo_postal;
char cidade[STRING_MAX];
}ST_MORADA;
```

Tabela 6: Representação da estrutura para a morada

Campo	Tipo	Descrição
rua	String	Nome da rua onde o cliente reside.
codigo_postal	Long Inteiro Positvo	Número referente ao código postal.
cidade	String	Nome da cidade do cliente.

3.4.3 Estrutura para Data

Da mesma forma, foi desenvolvida uma estrutura composta com informações relacionadas a data, como data de nascimento dos clientes ou data e hora das consultas. Foi designada de 'ST_DATA' e é constituída pelo dia, mês, ano e hora.

```
typedef struct {
unsigned int hora;
unsigned int dia;
unsigned int mes;
unsigned int ano;
}ST_DATA;
```

Tabela 7: Representação da estrutura para a data

Campo	Tipo	Descrição
hora	Inteiro Positivo	Número da hora.
dia	Inteiro Positivo	Número do dia.
mes	Inteiro Positivo	Número do mês.
ano	Inteiro Positivo	Número do ano.

4 Funções Implementadas

Neste capítulo, serão descritas as principais funções implementadas no projeto. O objetivo destas funções é permitir a inserção, alteração e consulta de dados relativos aos clientes, médicos e consultas, garantindo a organização e a eficiência do sistema.

4.1 Inserção de Dados

As funções de inserção de clientes, médicos e consultas, desempenham um papel crucial na adição de informações ao sistema, garantindo que as informações sejam organizadas, validadas e registadas.

4.1.1 Inserção de Clientes

A função permite adicionar um novo cliente, solicitando o nome do cliente, morada, data de nascimento, e-mail, número de identificação fiscal e número de utente. A função valida o limite máximo de clientes, garante a possibilidade de confirmar ou cancelar a inserção e armazena os dados num ficheiro externo.

```
void inserirClientes(ST_CLIENTE *clientes){
int opcao;
if(numeroClientes(clientes) >= MAX_CLIENTES){
printf("Número máximo de clientes atingido!\a\n");
delay(1);
return;
}else{
clear();
ST_CLIENTE cliente;
cliente.ID = numeroClientes(clientes) + 1;
printf("Nome do cliente: ");
fgets(cliente.nome, STRING_MAX, stdin);
cliente.nome[strcspn(cliente.nome, "\n")] = '\0';
obterMorada(&cliente);
printf("Rua: %s\n", cliente.morada.rua);
printf("Cidade: %s\n", cliente.morada.cidade);
```

```
        printf("Data de nascimento (dd-mm-aaaa): ");
        scanf("%2d-%2d-%4d", &cliente.data_nascimento.dia,
&cliente.data_nascimento.mes, &cliente.data_nascimento.ano);
        limparBuffer();
        printf("E-Mail: ");
        fgets(cliente.email, STRING_MAX, stdin);
        cliente.email[strcspn(cliente.email, "\n")] = '\0';
        printf("NIF: ");
        scanf("%9d", &cliente.NIF);
        limparBuffer();
        printf("SNS: ");
        scanf("%9d", &cliente.SNS);
        limparBuffer();
        cliente.estado = true;
        do {
        printf("Deseja confirmar inserção do cliente? [1] - SIM e [0] - NÃO: ");
        scanf("%1d", &opcao);
        limparBuffer();
        } while(opcao != 0 && opcao != 1);
        if (opcao){
        confirmarClientes(clientes, cliente);
        inserirFicheiroCliente(cliente);
        printf("Cliente inserido com sucesso.\n");
        delay(1);
        }
        }
        }
```

4.1.2 Inserção de Médicos

A função é responsável pela inserção de novos médicos, regista informações como nome e especialidade, tal como na inserção de clientes, a função faz a verificação do limite de médicos, confirmação da inserção e armazena os dados num ficheiro externo.

```
        void inserirMedicos(ST_MEDICO *medicos){
        int opcao;
        if(numeroMedicos(medicos) >= MAX_MEDICOS){
        printf("Número máximo de médicos atingido!\a\n");
        delay(1);
        return;
        }else {
        clear();
        ST_MEDICO medico;
        medico.ID = numeroMedicos(medicos) + 1;
        printf("Nome do médico: ");
```

```
fgets(medico.nome, STRING_MAX, stdin);
medico.nome[strcspn(medico.nome, "\n")] = '\0';
obterEspecialidade(&medico);
medico.estado = true;

do {
printf("Deseja confirmar inserção do médico? [1] - SIM e [0] - NÃO: ");
scanf("%1d", &opcao);
limparBuffer();
}while(opcao!= 0 && opcao != 1);
if(opcao){
confirmarMedicos(medicos, medico);
inserirFicheiroMedico(medico);
printf("Médico inserido com sucesso.\n");
delay(1);
}
}
}
```

4.1.3 Agendamento de Consultas

A função realiza o agendamento de consultas, associa a consulta a um cliente e médico já existente, verifica a disponibilidade do cliente e médico em um horário específico, de igual forma armazena as informações num ficheiro externo.

```
void agendarConsultas(ST_CONSULTA *consultas, ST_CLIENTE *clientes,
ST_MEDICO *medicos){
unsigned int clienteID, medicoID;
int opcao;
if(!numeroClientes(clientes)){
printf("Não existe clientes registados.\a\n");
delay(1);
return;
}
if (!numeroMedicos(medicos)){
printf("Não existe médicos registados.\a\n");
delay(1);
return;
}
clear();
printf("ID do cliente: ");
scanf("%d", &clienteID);
limparBuffer();
if (clienteID <= 0 || clienteID > numeroClientes(clientes)){
printf("O ID é inválido.\a\n");
```

```
        delay(1);
        return;
        }
        if(!clientes[clienteID - 1].estado){
        printf("O cliente está inativo.\a\n");
        delay(1);
        return;
        }
        printf("ID do médico: ");
        scanf("%d", &medicoID);
        if(medicoID <= 0 || medicoID > numeroMedicos(medicos)){
        printf("O ID é inválido.\a\n");
        delay(1);
        return;
        }
        if(!medicos[medicoID - 1].estado){
        printf("O médico está indisponível.\a\n");
        delay(1);
        return;
        }
        ST_CONSULTA consulta;
        consulta.ID = numeroConsultas(consultas) + 1;
        consulta.cliente = obterCliente(clientes, clienteID);
        consulta.medico = obterMedico(medicos, medicoID);
        do {
        printf("Data da consulta (dd-mm-aaaa): ");
        scanf("%2d-%2d-%4d", &consulta.data_inicial.dia, &consulta.data_inicial.mes,
&consulta.data_inicial.ano);
        limparBuffer();
        printf("Hora (8h - 18h): ");
        scanf("%2d", &consulta.data_inicial.hora);
        limparBuffer();
        if(!verificarDisponibilidade(consultas, &consulta)){
        printf("Horário não é válido.\a\n");
        delay(1);
        }
        }while(!verificarDisponibilidade(consultas, &consulta));
        printf("Data: %2d-%2d-%4d | Ás %2dh até %2dh\n", consulta.data_inicial.dia,
consulta.data_inicial.mes, consulta.data_inicial.ano, consulta.data_inicial.hora,
consulta.data_final.hora);
        consulta.estado = Agendado;
        do {
        printf("Deseja confirmar agendamento da consulta? [1] - SIM e [0] - NÃO: ");
        scanf("%d", &opcao);
        limparBuffer();
        }while(opcao != 0 && opcao != 1);
        if(opcao){
```

```
confirmarConsultas(consultas, consulta);
inserirFicheiroConsulta(consulta);
printf("Consulta agendada com sucesso.\n");
delay(1);
}
}
```

4.2 Alteração de Dados

As funções de alteração de dados garantem uma atualização e uma flexibilidade das informações armazenadas no sistema, mantendo a precisão e relevância dos registos.

4.2.1 Alterar Dados de Clientes

A função de alterar dados de clientes, solicita a identificação de um cliente, faz a verificação se o número fornecido está dentro do intervalo válido, exibe o menu de opções para alteração de diferentes campos do cliente e por fim atualiza os dados no ficheiro externo.

```
void alterarDadosClientes(ST_CLIENTE *clientes){
int ID, opcao;
clear();
printf("ID do cliente: ");
scanf("%d", &ID);
limparBuffer();
if (ID <= 0 || ID > numeroClientes(clientes)){
printf("O ID é inválido.\a\n");
delay(1);
return;
}
ST_CLIENTE *cliente = &clientes[ID - 1];
do {
clear();
printf("[1] - Nome\n");
printf("[2] - Código Postal\n");
printf("[3] - Data de nascimento\n");
printf("[4] - E-Mail\n");
printf("[5] - NIF\n");
printf("[6] - SNS\n");
printf("[0] - Sair\n");
printf("\n-> ");
scanf("%1d", &opcao);
```

```
        limparBuffer();
        switch(opcao){
        case 1:
        clear();
        printf("Nome: ");
        fgets(cliente->nome, STRING_MAX, stdin);
        cliente->nome[strcspn(cliente->nome, "\n")] = '\0';
        break;
        case 2:
        obterMorada(cliente);
        break;
        case 3:
        clear();
        printf("Data de nascimento (dd-mm-aaaa): ");
        scanf("%2d-%2d-%4d", &cliente->data_nascimento.dia, &cliente-
>data_nascimento.mes, &cliente->data_nascimento.ano);
        limparBuffer();
        break;
        case 4:
        clear();
        printf("E-Mail: ");
        fgets(cliente->email, STRING_MAX, stdin);
        cliente->email[strcspn(cliente->email, "\n")] = '\0';
        break;
        case 5:
        clear();
        printf("NIF: ");
        scanf("%9d", &cliente->NIF);
        limparBuffer();
        break;
        case 6:
        clear();
        printf("SNS: ");
        scanf("%9d", &cliente->SNS);
        limparBuffer();
        break;
        case 0:
        atualizarFicheiroCliente(clientes);
        break;
        default:
        printf("Opção não é válida.\a\n");
        delay(1);
        break;
        }
        }while (opcao != 0);
        }
```

4.2.2 Alterar Dados de Médicos

A função para alterar dados dos médicos, faz solicitação do número de identificação do médico, valida o número fornecido, exibe o menu com os campos disponíveis para alterar, como o nome e especialidade, por fim atualiza o ficheiro externo correspondente aos médicos.

```
void alterarDadosMedicos(ST_MEDICO *medicos){
int ID, opcao;
clear();
printf("ID do médico: ");
scanf("%d", &ID);
limparBuffer();
if(ID <= 0 || ID > numeroMedicos(medicos)){
printf("O ID é inválido.\a\n");
delay(1);
return;
}
ST_MEDICO *medico = &medicos[ID - 1];
do {
clear();
printf("[1] - Nome\n");
printf("[2] - Especialidade\n");
printf("[0] - Sair\n");
printf("\n-> ");
scanf("%1d", &opcao);
limparBuffer();
switch(opcao){
case 1:
clear();
printf("Nome: ");
fgets(medico->nome, STRING_MAX, stdin);
medico->nome[strcspn(medico->nome, "\n")] = '\0';
printf("Nome do médico alterado com sucesso.\n");
delay(1);
break;
case 2:
obterEspecialidade(medico);
printf("Especialidade alterada com sucesso.\n");
delay(1);
break;
case 0:
atualizarFicheiroMedico(medicos);
break;
default:
```

```
printf("Opção não é válida.\a\n");
delay(1);
break;
}
}while(opcao != 0);
}
```

4.2.3 Alterar Dados de Consultas

A função para atualizar dados das consultas, é implementada para modificar os dados, como alteração do cliente, médico, ou horário da consulta. É realizada a verificação de consultas registadas, solicitando depois o número de identificação da consulta, faz a verificação da consulta, exibe o menu com as opções para alteração de dados, por fim atualiza o ficheiro externo que armazena os dados das consultas.

```
void atualizarConsultas(ST_CONSULTA *consultas, ST_CLIENTE *clientes,
ST_MEDICO *medicos){
unsigned int ID, clienteID, medicoID;
int opcao;
if (!numeroConsultas(consultas)){
printf("Não há consultas registadas.\a\n");
delay(1);
return;
}
clear();
printf("ID da consulta: ");
scanf("%d", &ID);
limparBuffer();
ST_CONSULTA *consulta = obterConsulta(consultas, ID);
if (consulta == NULL){
printf("Consulta não foi encontrada.\a\n");
delay(1);
return;
}
if (consulta->estado == Realizado){
printf("Consulta já foi realizada.\n");
delay(1);
return;
}
if (consulta->estado == Cancelado){
printf("Consulta já foi cancelada.\a\n");
delay(1);
return;
```

```
        }
        do {
        clear();
        printf("[1] - ID do cliente\n");
        printf("[2] - ID do médico\n");
        printf("[3] - Horário da consulta\n");
        printf("[0] - Sair\n");
        printf("\n-> ");
        scanf("%1d", &opcao);
        limparBuffer();
        switch(opcao){
        case 1:
        do {
        clear();
        printf("ID do cliente: ");
        scanf("%d", &clienteID);
        limparBuffer();
        if(!clientes[clienteID - 1].estado){
        printf("O cliente está inativo.\a\n");
        delay(1);
        }
        }while(!clientes[clienteID - 1].estado);
        consulta->cliente = obterCliente(clientes, clienteID);
        break;
        case 2:
        do {
        clear();
        printf("ID do médico: ");
        scanf("%d", &medicoID);
        limparBuffer();
        if(!medicos[medicoID -1].estado){
        printf("O médico está indisponível.\a\n");
        delay(1);
        }
        }while(!medicos[medicoID -1].estado);
        consulta->medico = obterMedico(medicos, medicoID);
        break;
        case 3:
        do {
        clear();
        printf("Data da consulta (dd-mm-aaaa): ");
        scanf("%2d-%2d-%4d", &consulta->data_inicial.dia, &consulta-
>data_inicial.mes, &consulta->data_inicial.ano);
        limparBuffer();
        printf("Hora (8h - 18h): ");
        scanf("%2d", &consulta->data_inicial.hora);
        limparBuffer();
```

```
if(!verificarDisponibilidade(consultas, consulta)){
printf("Horário não é válido.\a\n");
delay(1);
}
}while(!verificarDisponibilidade(consultas, consulta));
infoConsultas(*consulta);
delay(5);
break;
case 0:
atualizarFicheiroConsulta(consultas);
break;
default:
printf("Opção não é válida.\a\n");
delay(1);
break;
}
}while(opcao != 0);
}
```

4.3 Consulta de Dados

As funções responsáveis pela consulta de informações, representam a capacidade do sistema de visualizar e procurar dados previamente inseridos. Essa funcionalidade é essencial em sistemas de gestão, pois permite aos utilizadores aceder as informações sobre clientes, médicos e consultas.

4.3.1 Listar Clientes

A função solicita ao utilizador o número de identificação do cliente, valida o número se está dentro do intervalo, localiza o cliente no array dos clientes utilizando o número fornecido ajustado para índice zero do array (ID - 1). Por fim é chamada a função para exibir informações dos clientes que recebe o número do índice do cliente, com um atraso de 10 segundos para o utilizador visualizar os dados antes de retornar ao função anterior.

```
void consultarDadosClientes(ST_CLIENTE *clientes){
int ID;
clear();
printf("ID do cliente: ");
scanf("%d", &ID);
```

```
limparBuffer();
if (ID <= 0 || ID > numeroClientes(clientes)){
printf("ID não é válido.\a\n");
delay(1);
return;
}
clear();
infoClientes(clientes[ID - 1]);
delay(10);
return;
}
```

4.3.2 Listar Médicos

A função para consultar informações dos médicos, solicita o número de identificação do médico, verifica se está dentro do intervalo válido, caso seja válido é chamada a função para exibir as informações recebendo o índice do médico. Após a exibição das informações, é aguardado 10 segundnos para permitir ao utilizar ler as informações.

```
void consultarDadosMedicos(ST_MEDICO *medicos){
int ID;
clear();
printf("ID do médico: ");
scanf("%d", &ID);
limparBuffer();
if(ID <= 0 || ID > numeroMedicos(medicos)){
printf("ID não é válido.\a\n");
delay(1);
return;
}
clear();
infoMedicos(medicos[ID - 1]);
delay(10);
return;
}
```

4.3.3 Consultar Histórico de Clientes

```
void obterHistoricoConsultasCliente(ST_CONSULTA *consultas, ST_CLIENTE
*clientes){
unsigned int clienteID, encontrados = 0;
if (!numeroConsultas(consultas)){
printf("Não há consultas registadas.\a\n");
delay(1);
return;
}
clear();
printf("ID do cliente: ");
scanf("%d", &clienteID);
limparBuffer();
if (!clientes[clienteID - 1].estado){
printf("O cliente está inativo.\a\n");
delay(1);
return;
}
clear();
for (int i = 0; i < numeroConsultas(consultas); i++){
if(consultas[i].cliente->ID == clienteID){
infoConsultas(consultas[i]);
encontrados++;
}
}
if (!encontrados){
printf("Não há consultas para o cliente.\a\n");
delay(1);
return;
}
delay(15);
return;
}
```

4.3.4 Consultar Consultas para Dia Atual por Médico

A função é responsável por exibir as consultas agendadas para um médico específico no dia atual. O funcionamento da função envolve a identificação do dia atual, utilizando outra função para obter a data corrente, verificação de registos de consultas, solicitação do número de identificação do médico e validação do mesmo, procura por consultas correspondentes e exibição das informações, caso não haja consultas correspondentes exibe uma mensagem.

```
	void obterListaConsultasDiaAtualMedico(ST_CONSULTA *consultas,
ST_MEDICO *medicos){
	ST_DATA data;
	dataAtual(&data);
	unsigned int medicoID, encontrados = 0;
	if (!numeroConsultas(consultas)){
	printf("Não há consultas registadas.\a\n");
	delay(1);
	return;
	}
	clear();
	printf("ID do médico: ");
	scanf("%d", &medicoID);
	limparBuffer();
	if(!medicos[medicoID -1].estado){
	printf("O médico está indisponível.\a\n");
	delay(1);
	return;
	}
	clear();
	for (int i = 0; i < numeroConsultas(consultas); i++){
	if(consultas[i].medico->ID == medicoID && consultas[i].data_inicial.ano ==
data.ano && consultas[i].data_inicial.mes == data.mes &&
consultas[i].data_inicial.dia == data.dia){
	infoConsultas(consultas[i]);
	encontrados++;
	}
	}
	if (!encontrados){
	printf("Não há consultas para o médico no dia de hoje.\a\n");
	delay(1);
	return;
	}
	delay(15);
	return;
	}
```

4.4 Gestão de Ficheiros

A gestão de ficheiros é algo que diferência este projeto desenvolvido, algo crucial em sistemas de gestão que necessitam de armazenar, recuperar e atualizar informações de forma persistente.

No contexto deste sistema, a funcionalidade de gestão de ficheiros foi implementada para assegurar que as informações inseridas durante a execução da aplicação possam ser armazenadas, consultadas e alteradas, mesmo após o encerramento da aplicação.

4.4.1 Inserir Dados em Ficheiros

A função de inserir dados nos ficheiros é fundamental para armazenar as informações relevantes da aplicação. Esta funcionalidade permite armazenar os dados dos clientes, médicos e consultas, garantindo que eles possam ser acessados posteriormente, mesmo após o encerramento da aplicação.

O processo de inserção de dados é realizado por a função para abrir o ficheiro correspondente em modo de anexação, de forma a evitar a sobrescrição de dados existentes. Após a abertura do ficheiros, os dados são formatados e gravados utilizando a função fprintf.

```
void inserirFicheiroCliente(ST_CLIENTE cliente){
FILE *ficheiro = fopen("data/clientes.txt", "a");
if(ficheiro == NULL){
printf("Erro.\n");
return;
}
fprintf(ficheiro, "%d,%s,%lu,%s,%s,%2d,%2d,%4d,%s,%9d,%9d,%s\n", cliente.ID,
cliente.nome, cliente.morada.codigo_postal, cliente.morada.rua,
cliente.morada.cidade, cliente.data_nascimento.dia, cliente.data_nascimento.mes,
cliente.data_nascimento.ano, cliente.email, cliente.NIF, cliente.SNS, cliente.estado ?
"Ativo" : "Inativo");
fclose(ficheiro);
return;
}
```

```
        void inserirFicheiroMedico(ST_MEDICO medico){
        FILE *ficheiro = fopen("data/medicos.txt", "a");
        if(ficheiro == NULL){
        printf("Erro.\n");
        return;
        }
        fprintf(ficheiro, "%d,%s,%s,%s\n", medico.ID, medico.nome,
medico.especialidade, medico.estado ? "Disponível" : "Indisponível");
        fclose(ficheiro);
        return;
        }

        void inserirFicheiroConsulta(ST_CONSULTA consulta){
        char *estadoConsulta[3] = { "Cancelado", "Agendado", "Realizado"};
        FILE *ficheiro = fopen("data/consultas.txt", "a");
        if(ficheiro == NULL){
        printf("Erro.\n");
        return;
        }
        fprintf(ficheiro, "%d,%d,%s,%d,%s,%s,%2d,%2d,%4d,%2d,%2d,%s\n", consulta.ID,
consulta.cliente->ID, consulta.cliente->nome, consulta.medico->ID,
consulta.medico->nome, consulta.medico->especialidade, consulta.data_inicial.dia,
consulta.data_inicial.mes, consulta.data_inicial.ano, consulta.data_inicial.hora,
consulta.data_final.hora, estadoConsulta[consulta.estado]);
        fclose(ficheiro);
        return;
        }
```

4.4.2 Carregar Dados dos Ficheiros

As funções de leitura de dados a partir dos ficheiros são responsáveis pela recuperação dos dados anteriormente inseridos. Esta funcionalidade evita a perda de informação e assegura a continuidade das operações em diferentes execuções do programa.

A utilização de funções como fgets e strtok garante a manipulação eficaz das linhas e dos campos dos ficheiros, possibilitando uma leitura ordenada, estruturada e segura de grandes volumes de dados.

```
void carregarFicheiroCliente(ST_CLIENTE *clientes){
char linha[1024], *token;
int i = 0;
FILE *ficheiro;
ficheiro = fopen("data/clientes.txt", "r");
if(ficheiro == NULL){
return;
}
while(fgets(linha, sizeof(linha), ficheiro) && i < MAX_CLIENTES){
linha[strcspn(linha, "\n")] = '\0';
token = strtok(linha, ",");
clientes[i].ID = (atoi(token));
token = strtok(NULL, ",");
strncpy(clientes[i].nome, token, STRING_MAX);
token = strtok(NULL, ",");
clientes[i].morada.codigo_postal = (atoi(token));
token = strtok(NULL, ",");
strncpy(clientes[i].morada.rua, token, STRING_MAX);
token = strtok(NULL, ",");
strncpy(clientes[i].morada.cidade, token, STRING_MAX);
token = strtok(NULL, ",");
clientes[i].data_nascimento.dia = (atoi(token));
token = strtok(NULL, ",");
clientes[i].data_nascimento.mes = (atoi(token));
token = strtok(NULL, ",");
clientes[i].data_nascimento.ano = (atoi(token));
token = strtok(NULL, ",");
strncpy(clientes[i].email, token, STRING_MAX);
token = strtok(NULL, ",");
clientes[i].NIF = (atoi(token));
token = strtok(NULL, ",");
clientes[i].SNS = (atoi(token));
token = strtok(NULL, ",");
clientes[i].estado = (strcmp(token, "Ativo") == 0);
i++;
}
fclose(ficheiro);
return;
}


void carregarFicheiroMedico(ST_MEDICO *medicos){
char linha[1024], *token;
int i = 0;
FILE *ficheiro;
ficheiro = fopen("data/medicos.txt", "r");
if (ficheiro == NULL){
```

```
        printf("Erro.\n");
        return;
        }
        while(fgets(linha, sizeof(linha), ficheiro) && i < MAX_MEDICOS){
        linha[strcspn(linha, "\n")] = '\0';
        token = strtok(linha, ",");
        medicos[i].ID = (atoi(token));
        token = strtok(NULL, ",");
        strncpy(medicos[i].nome, token, STRING_MAX);
        token = strtok(NULL, ",");
        strncpy(medicos[i].especialidade, token, STRING_MAX);
        token = strtok(NULL, ",");
        medicos[i].estado = (strcmp(token, "Disponível") == 0);
        i++;
        }
        fclose(ficheiro);
        return;
        }

        void carregarFicheiroConsulta(ST_CONSULTA *consultas, ST_CLIENTE
*clientes, ST_MEDICO *medicos){
        char linha[1024], *token;
        unsigned int clienteID, medicoID;
        int i = 0;
        FILE *ficheiro;
        ficheiro = fopen("data/consultas.txt", "r");
        if (ficheiro == NULL){
        printf("Erro\n");
        return;
        }
        while(fgets(linha, sizeof(linha), ficheiro) && i < MAX_CONSULTAS){
        linha[strcspn(linha, "\n")] = '\0';
        token = strtok(linha, ",");
        consultas[i].ID = (atoi(token));
        token = strtok(NULL, ",");
        clienteID = (atoi(token));
        consultas[i].cliente = obterCliente(clientes, clienteID);
        token = strtok(NULL, ",");
        token = strtok(NULL, ",");
        medicoID = (atoi(token));
        consultas[i].medico = obterMedico(medicos, medicoID);
        token = strtok(NULL, ",");
        token = strtok(NULL, ",");
        token = strtok(NULL, ",");
        consultas[i].data_inicial.dia = (atoi(token));
        token = strtok(NULL, ",");
        consultas[i].data_inicial.mes = (atoi(token));
```

```
token = strtok(NULL, ",");
consultas[i].data_inicial.ano = (atoi(token));
token = strtok(NULL, ",");
consultas[i].data_inicial.hora = (atoi(token));
token = strtok(NULL, ",");
consultas[i].data_final.hora = (atoi(token));
token = strtok(NULL, ",");
if(strcmp(token, "Cancelado") == 0){
consultas[i].estado = 0;
}else if(strcmp(token, "Agendado") == 0){
consultas[i].estado = 1;
}else if(strcmp(token, "Realizado") == 0){
consultas[i].estado = 2;
}
consultas[i].data_final.dia = consultas[i].data_inicial.dia;
consultas[i].data_final.mes = consultas[i].data_inicial.mes;
consultas[i].data_final.ano = consultas[i].data_inicial.ano;
i++;
}
fclose(ficheiro);
return;
}
```

As funções correspondentes são invocadas logo após a definição das estruturas principais, assegurando que o utilizador ao entrar no menu principal, as informações pertinentes já estejam carregadas no sistema. No fim das funções, a aplicação apresenta o menu principal, que utiliza as informações carregadas para executar as operações desejadas.

4.4.3 Atualizar Dados dos Ficheiros

Para garantir que todas as alterações realizadas durante a execução da aplicação sejam persistidas, é essencial atualizar os ficheiros no momento adequado. As seguintes funções desempenham o papel de atualizar os dados atuais das estruturas nos ficheiros correspondentes, permitindo que as alterações sejam mantidas entre sessões.

Essa funcionalidade é essencial para manter o sistema e os dados atualizados, proporcionando uma experiência continua ao utilizador.

```
void atualizarFicheiroCliente(ST_CLIENTE *clientes){
FILE *ficheiro;
ficheiro = fopen("data/clientes.txt", "w");
if (ficheiro == NULL){
printf("Erro.\n");
return;
}
for (int i = 0; i < numeroClientes(clientes); i++){
fprintf(ficheiro, "%d,%s,%lu,%s,%s,%2d,%2d,%4d,%s,%9d,%9d,%s\n", clientes[i].ID,
clientes[i].nome, clientes[i].morada.codigo_postal, clientes[i].morada.rua,
clientes[i].morada.cidade, clientes[i].data_nascimento.dia,
clientes[i].data_nascimento.mes, clientes[i].data_nascimento.ano, clientes[i].email,
clientes[i].NIF, clientes[i].SNS, clientes[i].estado ? "Ativo" : "Inativo");
}
fclose(ficheiro);
return;
}
void atualizarFicheiroMedico(ST_MEDICO *medicos){
FILE *ficheiro;
ficheiro = fopen("data/medicos.txt", "w");
if (ficheiro == NULL){
printf("Erro.\n");
return;
}
for (int i = 0; i < numeroMedicos(medicos); i++){
fprintf(ficheiro, "%d,%s,%s,%s\n", medicos[i].ID, medicos[i].nome,
medicos[i].especialidade, medicos[i].estado ? "Disponível" : "Indisponível");
}
fclose(ficheiro);
return;
}
void atualizarFicheiroConsulta(ST_CONSULTA *consultas){
char *estadoConsulta[3] = {"Cancelado", "Agendado", "Realizado"};
FILE *ficheiro;
ficheiro = fopen("data/consultas.txt", "w");
if (ficheiro == NULL){
printf("Erro.\n");
return;
}
for (int i = 0; i < numeroConsultas(consultas); i++){
fprintf(ficheiro, "%d,%s,%s,%s,%2d,%2d,%4d,%2d,%2d,%s\n", consultas[i].ID,
consultas[i].cliente->nome, consultas[i].medico->nome, consultas[i].medico-
```

```
>especialidade, consultas[i].data_inicial.dia, consultas[i].data_inicial.mes,
consultas[i].data_inicial.ano, consultas[i].data_inicial.hora,
consultas[i].data_final.hora, estadoConsulta[consultas[i].estado]);
        }
        fclose(ficheiro);
        return;
        }
```

4.4.4 Considerações Finais

Estas funções foram desenvolvidas para operar de forma segura e eficiente, minimizando o risco de perda de dados e garantindo a confiabilidade do sistema, é crucial para sistemas que precisam de preservar informações, como registo de clientes, médicos e consultas.

Os ficheiros em formato de texto, podem ser facilmente utilizados por outras aplicações, promovendo a integração com sistemas externos, como ferramentas de análise de dados ou softwares de backup.

A gestão eficiente de ficheiros não apenas sustenta a operação interna da aplicação, mas também reforça a sua usabilidade e segurança, criando um sistema de gestão estável e pronto para suportar a evolução das suas funcionalidades.

4.5 Funções Auxiliares

As funções auxiliares desempenham um papel importante na organização e reutilização do código, proporcionam uma maior clareza e eficiência no desenvolvimento da aplicação. As funções implementadas são projetadas para realizar tarefas específicas e recorrentes que completam o funcionamento da aplicação, simplificando a lógica e legibilidade do código. Nesta secção são apresentadas funcionalidades como limpeza do buffer, delays para melhor interação do utilizador, manipulação de datas e horas, obtenção da cidade e da rua de acordo com o código postal inserido pelo utilizador, e obtenção da especialidade dos médicos. No presente subcapítulo é detalhado cada uma das funções auxiliares implementadas, destacando a sua utilidade.

4.5.1 Limpar o stdout

A função de limpar o stdout foi implementada para proporcionar ao utilizador uma interação mais organizada e eficiente com a aplicação. Essa funcionalidade limpa o output do terminal, removendo as informações desnecessárias ou já processadas, para manter o foco nas informações ou opções relevantes a cada etapa. A implementação utiliza as diretivas de compilação condicionais para garantir compatibilidade com sistemas Windows e sistemas Unix (Linux e MacOS).

```
void clear(void){
#ifdef _WIN32
system("cls");
#else
system("clear");
#endif
}
```

4.5.2 Pausa Temporária

A função para delays foi desenvolvida para introduzir uma pausa temporária na execução do programa, permitindo criar intervalos entre operações. De igual forma para garantir compatibilidade em diferentes sistemas operativos foram utilizadas diretivas de compilação, garantindo em sistemas Windows a utilização da função Sleep e em sistemas Unix (Linux e MacOS) a utilização da função sleep.

```
void delay(int num){
#ifdef _WIN32
Sleep(num * 1000);
#else
sleep(num);
#endif
}
```

4.5.3 Limpar o Buffer do stdin

Esta funcionalidade foi criada para garantir que o buffer do stdin seja totalmente limpo, removendo quaisquer caracteres, incluido o Newline (\n). Isso previne compartamentos indesejados como leitura incorreta dos dados seguintes. A função utiliza um loop while para consumir todos os caracteres até encontrar o Newline (\n) ou end-of-file (EOF).

```
void limparBuffer(void){
int c;
while((c = getchar()) != '\n' && c != EOF);
}
```

4.5.4 Obter Data Atual

A função foi desenvolvida para obter data e hora atuais do sistema e armazená-las em um estrutura do tipo ST_DATA. Utiliza as funções padrão da biblioteca time.h, os valores do dia atual é convertido para um estrutura tm, que fornece as informações separadas por ano, mês, dia, hora e minuto. Os valores então são atribuidos aos campos da estrutura, ajustando o mês e ano conforme o necessário.

```
void dataAtual(ST_DATA *data_hora_atual){
time_t t = time(NULL);
struct tm tm = *localtime(&t);
data_hora_atual->hora = tm.tm_hour;
data_hora_atual->dia = tm.tm_mday;
data_hora_atual->mes = tm.tm_mon + 1;
data_hora_atual->ano = tm.tm_year + 1900;
}
```

4.5.5 Obter Morada com o Código Postal

Como mencionado durante a apresentação do trabalho prático nº1, a tentativa de implementar uma função para obter a morada com base no código postal foi discutida com os professores.

A função para obter informações da morada com base no código postal inserido pelo utilizador é das mais únicas e elaboradas neste trabalho prático, foi projetada para associar a morada completa do cliente com base no código postal fornecido pelo utilizador. A implementação e o desenvolvimento exigiu a obtenção de um ficheiro de texto contendo uma base de dados de ruas, cidades e códigos postais de Portugal, disponível no site dos CTT. Antes de integrar esta função na aplicação principal, foi implementada e amplamente testada por meio de um script específicio localizado na pasta tests, demonstrando a eficácia da abordagem antes da implementação e utilização final.

A função faz a leitura do código postal introduzido pelo utilizador e, em seguinda, percorre o ficheiro com os códigos postais linha a

linha, separando os campos utilizando a função strtok para dividir a linha por tokens. Se for encontrado o código postal, as informações correspondentes da Rua e Cidade são armazenadas no respetivo cliente. Caso o código postal não seja encontrada, a aplicação exibe a mensagem, sendo necessário e altamente recomendado proceder a inserção ou modificação manualmente dos dados do cliente. Isso é fundamental para garantir que, em sessões seguintes, não ocorra erros ao carregar informações dos ficheiros dos clientes.

```
void obterMorada(ST_CLIENTE *cliente){
FILE *ficheiro;
char linha[1024], *token;
int encontrados = 0;
unsigned long int codigo_postal;
ficheiro = fopen("data/codigos_postais.txt", "r");
if (ficheiro == NULL){
printf("Erro\n");
return;
}
printf("Código Postal (1234567): ");
scanf("%7lu", &codigo_postal);
limparBuffer();
while (fgets(linha, sizeof(linha), ficheiro)){
token = strtok(linha, ",");
char *tokens[20];
int total_tokens = 0;
while(token != NULL){
tokens[total_tokens] = token;
total_tokens++;
token = strtok(NULL, ",");
}
if (total_tokens >= 3 && tokens[total_tokens - 3] != NULL && tokens[total_tokens -
2] != NULL){
unsigned long int cod_postal = (atoi(tokens[total_tokens - 3]) * 1000) +
atoi(tokens[total_tokens - 2]);
if(cod_postal == codigo_postal){
encontrados++;
strcpy(cliente->morada.rua, tokens[3]);
cliente->morada.rua[strcspn(cliente->morada.rua, "\n")] = '\0';
strcpy(cliente->morada.cidade, tokens[total_tokens - 1]);
cliente->morada.cidade[strcspn(cliente->morada.cidade, "\n")] = '\0';
cliente->morada.codigo_postal = codigo_postal;
break;
}
```

```
}
}
if(!encontrados){
printf("Código postal não encontrado.\a\n");
}
fclose(ficheiro);
return;
}
```

4.5.6 Obter Especialidade

A função para obter especialidade permite ao utilizador inserir e validar a especialidade de um médico a partir de uma lista já existente nos ficheiros. É realizada a leitura das especialidades de um ficheiro de texto, oferece duas opções ao utilizador, uma para visualizar a lista das especialidades disponíveis e outra inserir manualmente a especialidade. Caso o utilizador selecione a função para inserir especialidade, a função faz a verificação da especialidade e atribuí ao respetivo médico.

```
void obterEspecialidade(ST_MEDICO *medico){
FILE *ficheiro;
char linha[20], especialidade[20];
int opcao;
ficheiro = fopen("data/especialidade.txt", "r");
if (ficheiro == NULL){
printf("Erro\n");
return;
}
clear();
do {
clear();
printf("[1] - Obter uma lista das especialidades\n");
printf("[2] - Inserir a especialidade do médico\n");
printf("\n-> ");
scanf("%d", &opcao);
limparBuffer();
switch (opcao){
case 1:
listarEspecialidades();
break;
case 2:
clear();
int especialidade_valida = 0;
```

```
do{
printf("Especialidade: ");
fgets(especialidade, sizeof(especialidade), stdin);
especialidade[strcspn(especialidade, "\n")] = '\0';
while (fgets(linha, sizeof(linha), ficheiro) != NULL){
linha[strcspn(linha, "\n")] = '\0';
if(strncmp(especialidade, linha, 4) == 0){
strcpy(medico->especialidade, especialidade);
especialidade_valida = 1;
break;
}
}
if (!especialidade_valida){
printf("Especialidade não é válida.\a\n");
}
}while(especialidade_valida != 1);
break;
default:
printf("Opção inválida!\a\n");
break;
}
}while(opcao != 2);
fclose(ficheiro);
return;
}
```

4.5.7 Listar Especialidades

A função serve para listar as especialidades disponíveis, que estão armazenadas em um ficheiros de texto na pasta data. A função faz a leitura do ficheiro linha por linha e exibe cada especialidade no stdout.

```
void listarEspecialidades(void){
FILE *ficheiro;
char linha[20];
ficheiro = fopen("data/especialidade.txt", "r");
if(ficheiro == NULL){
printf("Erro.\n");
return;
}
clear();
printf("Lista das Especialidades Disponíveis:\n\n");
while(fgets(linha, sizeof(linha), ficheiro)){
printf("%s", linha);
}
```

```
fclose(ficheiro);
delay(10);
return;
}
```

4.5.8 Pressionar para Continuar

A função exibe uma mensagem e aguarda o utilizar inserir a tecla Enter, estando num loop while até o caracter de Newline (\n) seja detectado.

```
void pressionarEnter(void){
printf("Pressionar Enter para sair.\n");
while (getchar() != '\n');
}
```

5 Menus Interativos

Os menus são uma parte fundamental da fácil navegação em aplicações CLI (Command-line interface). Permitindo aos utilizadores interagir com o sistema de maneira intuitiva, oferecendo uma lista de opções que podem ser selecionadas para realizar diferentes operações.

No contexto deste trabalho prático, foram desenvolvidos menus que oferecem ao utilizador uma maneira simples de aceder a funcionalidades específicas, como obter informações dos clientes, médicos e consultas.

5.1 Menu Principal

O menu principal exibe as opções para aceder aos submenus seguintes, dos clientes, médicos e consultas. O loop continua até que seja inserido o valor zero, para sair da aplicação.

```
void menuPrincipal(ST_CLIENTE *clientes, ST_MEDICO *medicos, ST_CONSULTA
*consultas){
int opcao;
do {
clear();
printf("[1] - Gestão de Clientes\n");
printf("[2] - Gestão de Médicos\n");
printf("[3] - Gestão de Consultas\n");
printf("[0] - Sair\n");
printf("\n-> ");
scanf("%1d", &opcao);
limparBuffer();
switch (opcao){
case 1:
menuClientes(clientes);
break;
case 2:
menuMedicos(medicos);
break;
case 3:
menuConsultas(consultas, clientes, medicos);
break;
```

```
case 0:
break;
default:
printf("Opção não é válida!\a\n");
delay(1);
break;
}
} while (opcao != 0);
}
```

5.2 Menu dos Clientes

O submenu dos clientes, apresenta a gestão de informações relevantes aos clientes, permitindo a inserção, alteração, ativação e desativação, consulta e procura de clientes.

```
void menuClientes(ST_CLIENTE *clientes){
int opcao;
do {
clear();
printf("[1] - Inserir novo cliente\n");
printf("[2] - Alterar dados de um cliente\n");
printf("[3] - Ativar ou desativar um cliente\n");
printf("[4] - Consultar dados de um cliente\n");
printf("[5] - Obter uma lista de clientes ativos\n");
printf("[6] - Procurar um cliente pelo nome\n");
printf("[0] - Sair\n");
printf("\n-> ");
scanf("%1d", &opcao);
limparBuffer();
switch(opcao){
case 1:
inserirClientes(clientes);
break;
case 2:
alterarDadosClientes(clientes);
break;
case 3:
ativarDesativarClientes(clientes);
break;
case 4:
consultarDadosClientes(clientes);
break;
case 5:
obterListaClientesAtivos(clientes);
break;
```

```
case 6:
procurarClientesNome(clientes);
case 0:
break;
default:
printf("Opção não é válida!\a\n");
delay(1);
break;
}
}while(opcao != 0);
}
```

5.3 Menu dos Médicos

O submenu dos médicos exibe a gestão as informações dos médicos, permitindo a inserção, alteração, ativação ou desativação, consulta de dados e listagem de médicos ou especialidades.

```
void menuMedicos(ST_MEDICO *medicos){
int opcao;
do {
clear();
printf("[1] - Inserir novo médico\n");
printf("[2] - Alterar dados de um médico\n");
printf("[3] - Ativar ou desativar um médico\n");
printf("[4] - Consultar dados de um médico\n");
printf("[5] - Obter uma lista de todos os médicos\n");
printf("[6] - Obter uma lista de médicos disponíveis\n");
printf("[7] - Obter uma lista de médicos por especialidade\n");
printf("[8] - Procurar um médico pelo nome\n");
printf("[0] - Sair\n");
printf("\n-> ");
scanf("%1d", &opcao);
limparBuffer();
switch (opcao){
case 1:
inserirMedicos(medicos);
break;
case 2:
alterarDadosMedicos(medicos);
break;
case 3:
ativarDesativarMedicos(medicos);
break;
case 4:
consultarDadosMedicos(medicos);
```

```
break;
case 5:
obterListaTodosMedicos(medicos);
break;
case 6:
obterListaMedicosAtivos(medicos);
break;
case 7:
obterListaMedicosEspecialidade(medicos);
break;
case 8:
procurarMedicosNome(medicos);
break;
case 0:
break;
default:
printf("Opção não é válida!\a\n");
delay(1);
break;
}
}while(opcao != 0);
}
```

5.4 Menu das Consultas

O submenu da gestão de consultas, permitindo o agendamento, desmarção, atualização ou marcação de consultas como realizadas. Também oferece opções para obter lista de consultas do dia atual ou histórico de consultas de clientes.

```
void menuConsultas(ST_CONSULTA *consultas, ST_CLIENTE *clientes, ST_MEDICO
*medicos){
int opcao;
do {
clear();
printf("[1] - Agendar uma consulta\n");
printf("[2] - Desmarcar uma consulta\n");
printf("[3] - Marcar consulta como realizada\n");
printf("[4] - Atualizar uma consulta\n");
printf("[5] - Obter lista de consultas para o dia atual por médico\n");
printf("[6] - Obter histórico de consultas por cliente\n");
printf("[0] - Sair\n");
printf("\n-> ");
scanf("%1d", &opcao);
limparBuffer();
switch(opcao){
```

```
case 1:
agendarConsultas(consultas, clientes, medicos);
break;
case 2:
desmarcarConsultas(consultas);
break;
case 3:
marcarConsultasRealizadas(consultas);
break;
case 4:
atualizarConsultas(consultas, clientes, medicos);
break;
case 5:
obterListaConsultasDiaAtualMedico(consultas, medicos);
break;
case 6:
obterHistoricoConsultasCliente(consultas, clientes);
break;
case 0:
break;
default:
printf("Opção não é válida!\a\n");
delay(1);
break;
}
}while(opcao != 0);
}
```

6 Fluxogramas

Para facilitar o entendimento da aplicação, foi ainda realizado alguns fluxogramas que ilustram os principais processos e fluxos de dados dentro do sistema. Dada a complexidade e o grande volume de linhas de código da aplicação, não foi possível criar fluxogramas para todas as funcionalidades. No entanto, os fluxogramas existentes fornecem uma visão clara dos fluxos essenciais, ajudando a compreensão do comportamento da aplicação e no processo de tomada de decisão em diversas operações.

Esses fluxogramas servem como uma ferramenta útil para quem precisa de entender rapidamente como a aplicação lida com diferentes situações, desde a gestão de clientes e médicos até ao agendamento de consultas, e são fundamentais para documentar visualmente as funcionalidades mais complexas.

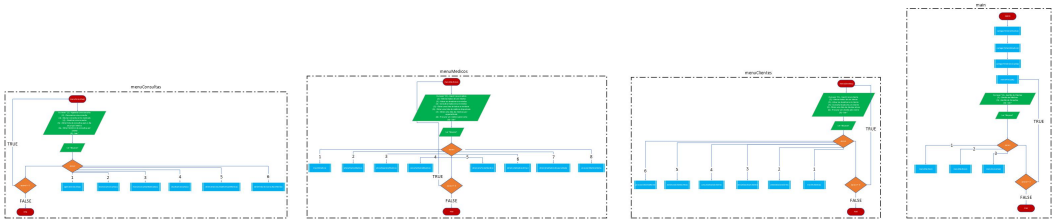

Figura 1: Fluxograma do menu.c

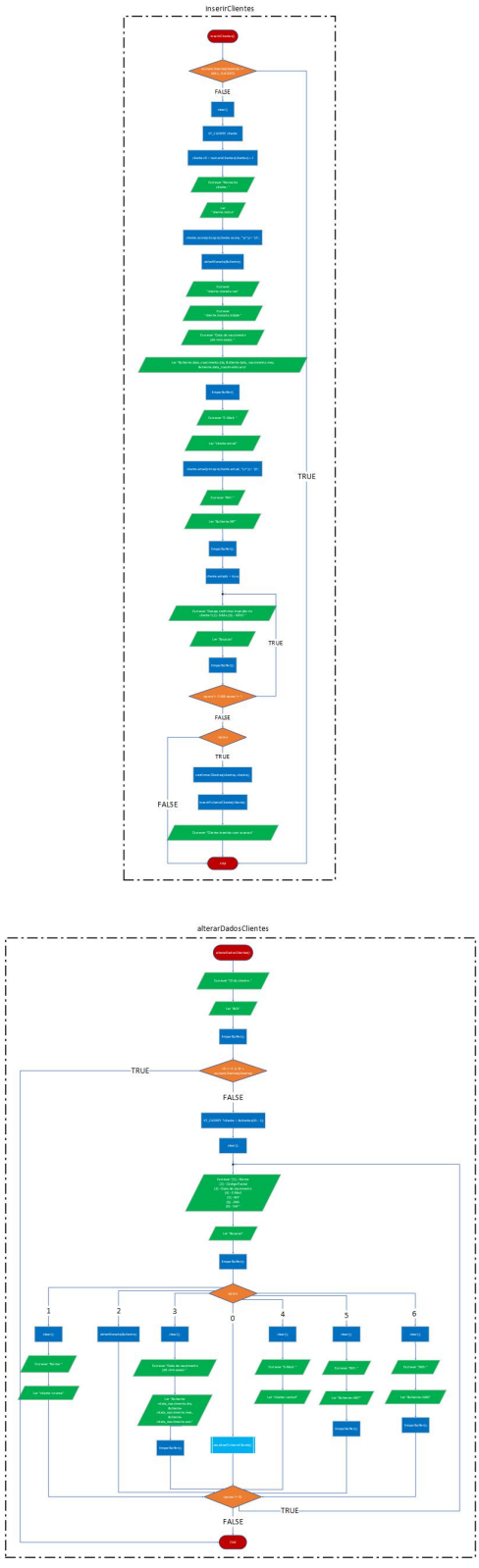

Figura 2: Fluxograma do cliente.c

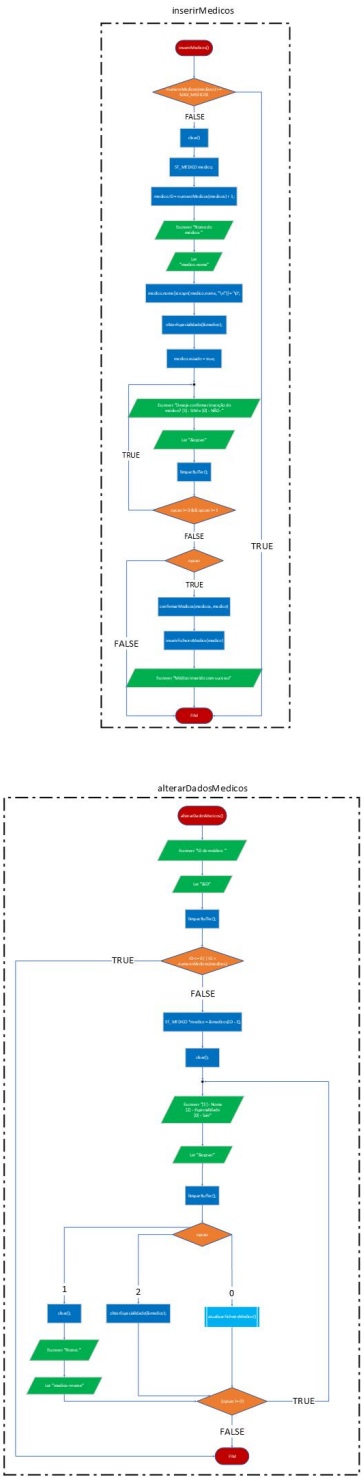

Figura 3: Fluxograma do medico.c

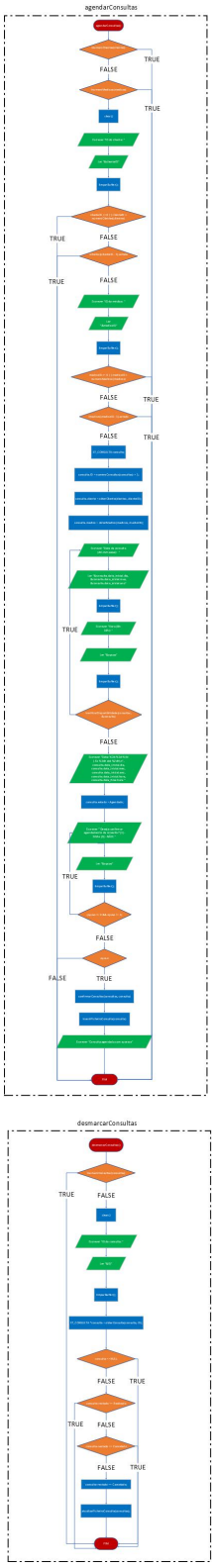

Figura 4: Fluxograma do consulta.c

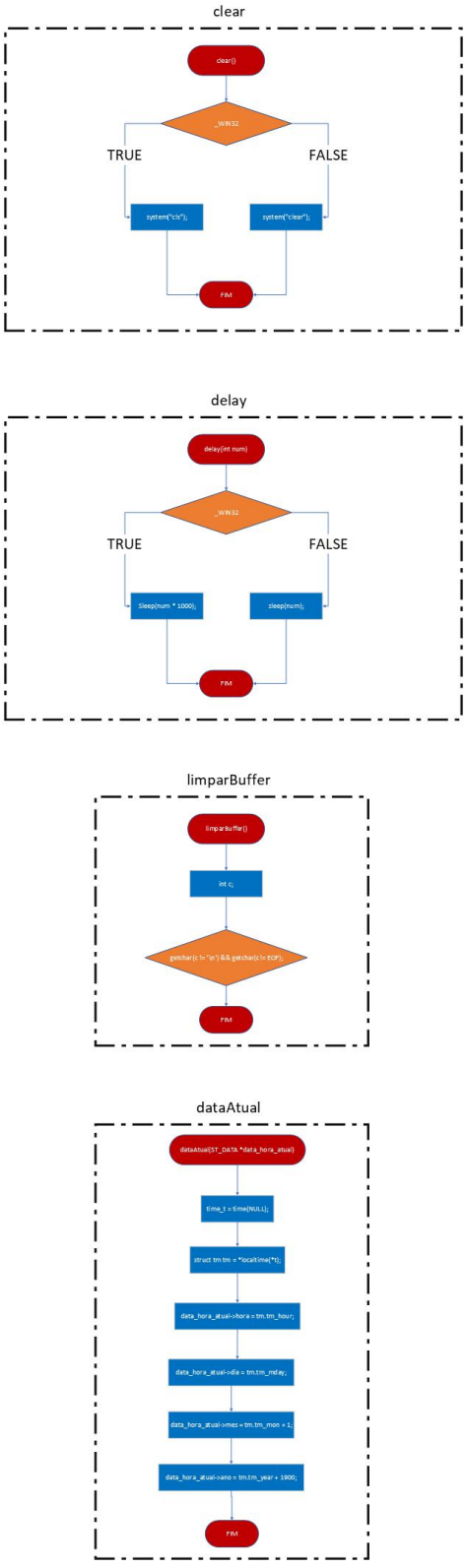

Figura 5: Fluxograma do auxiliares.c

7 Conclusão

O projeto da aplicação de gestão para clínicas, desenvolvido em linguagem C, foi concluído atendendo aos requisitos propostos no enunciado. Uma solução abrangente para gerenciar eficientemente os recursos de uma clínica, permitindo a gestão dos clientes, médicos e consultas, oferecendo uma interface simples e funcional.

O projeto utiliza conceitos como vetores estáticos, manipulação de ficheiros e estruturas, aponstadores e programação modular. Oferece uma implementação organizada e de fácil acesso aos dados. Além da utilização de programação por módulos, dividindo o projeto em diversos módulos, cada um responsável por funções específicas, facilitando a processo de testes e correção de erros.

Foi dada atenção especial à implementação de funcionalidades críticas, como a obtenção de moradas a partir de códigos postais e a verificação de especialidades dos médicos. A criação de fluxogramas foi também desenvolvida, proporcionando uma visão mais clara do funcionamento dos processos da aplicação.

Embora a aplicação seja funcional para o seu uso, a implementação de melhorias adicionais trairia vantagens ao projeto, a utilização de estruturas de dados avançadas como conceitos de fila, pilha ou árvores binárias, e a utilização de memória dinâmica para o gerenciamento de memória conforme necessário, oferecem um bom caminho para expandir o conhecimento e otimizar o projeto.

Em resumo, o projeto de gestão de clientes, médicos e consultas, além de funcional e simples, serve como uma boa base para futuros projetos, evidenciando a organização do código como essencial para o desenvolvimento e uma possível adição futura de funcionalidades avançadas.

8 Bibliografia

Damas, L. (2015). Linguagem C (24th ed.). Lisboa: FCA.