RELATÓRIO

[redacted]

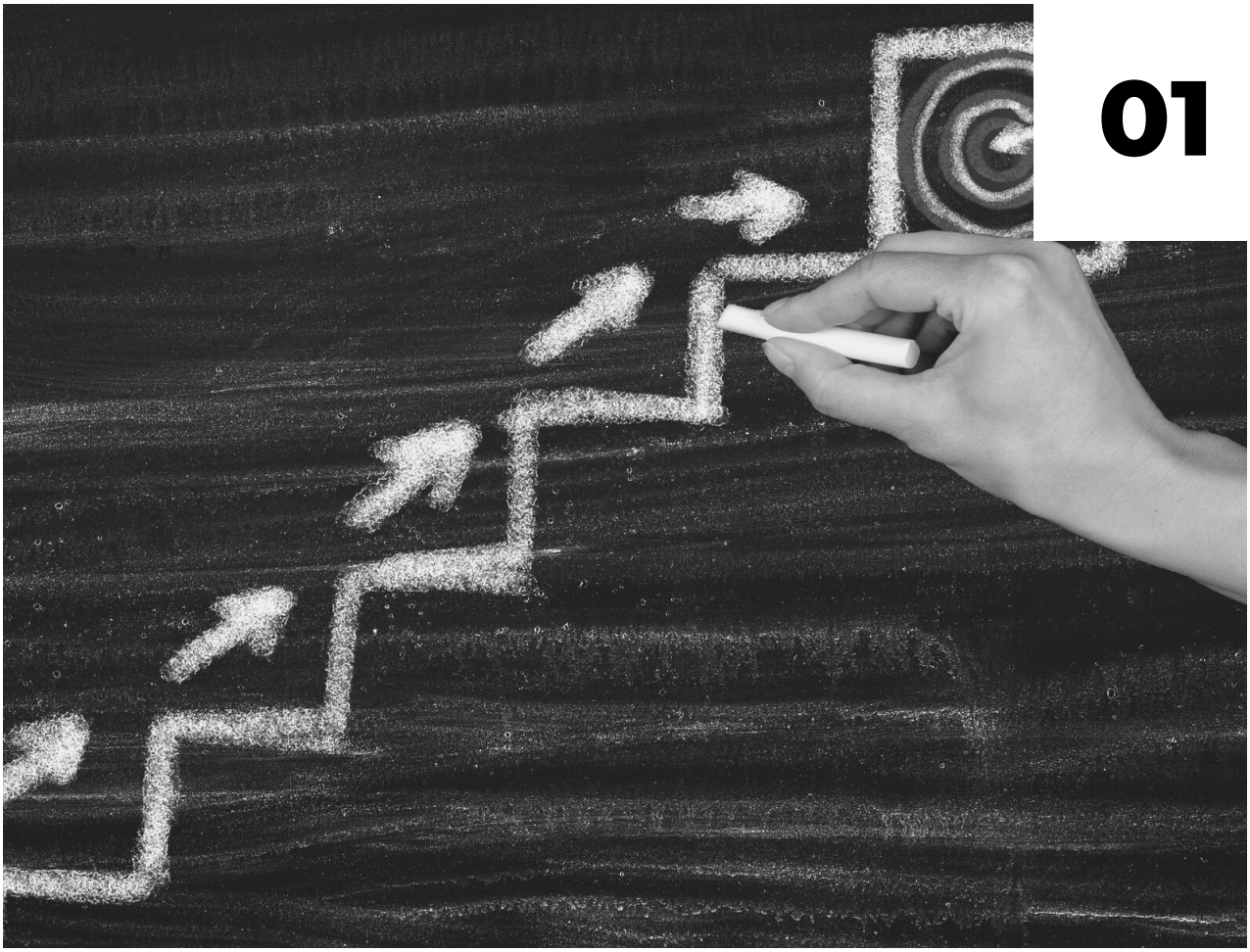
UNIVERSIDADE D COIMBRA FACULDADE DE CIÊNCIAS E TECNOLOGIA

MANUAL DO PROGRAMADOR

[redacted]

Carolina Reis, nº 2022220606
Tiago Silva, nº 2022211489

Data: 31 de maio de 2023
Docente: David Abreu
Unidade Curricular:
Princípios de Programação Procedimental



01

INTRODUÇÃO

A gestão eficiente de negócios é um requisito fundamental, que visa assegurar a fluidez dos serviços prestados, bem como a qualidade dos mesmos. Neste contexto, a utilização de ferramentas tecnológicas pode desempenhar um papel crucial, ao simplificar e agilizar muitos processos. Com esta perspetiva em mente, desenvolvemos um programa em linguagem C, cujo objetivo é auxiliar os funcionários de uma oficina na administração das suas reservas e pré-reservas (reservas em fila de espera).

O seguinte **Manual do Programador** fornece uma visão geral do sistema de gestão de oficinas implementado, abordando os seguintes tópicos: i) estruturas de dados em C; ii) estruturação do código; iii) processo de desenvolvimento de *software* e iv) precauções e boas práticas.

Podem também ser esperados, ao longo do relatório, alguns detalhes técnicos relevantes e diretrizes de programação, relacionadas com o processo de desenvolvimento.

ESTRUTURAS DE DADOS EM C

02

Eficiência e organização: O papel das filas de espera e das listas ligadas

Listas ligadas para armazenar reservas

As listas ligadas (ou *linked lists*) proporcionam uma maior flexibilidade na manipulação dos elementos, sendo compostas por nós encadeados que contém um valor e um ponteiro para o próximo nó. Além de permitirem um armazenamento dinâmico (reservas podem ser feitas e canceladas a qualquer instante), possibilitam a inserção e remoção eficiente de elementos em qualquer posição. Conclui-se, portanto, que as listas ligadas são uma escolha adequada para armazenar reservas, pelo acesso rápido e capacidade de manipulação individual. Tudo isto permite uma gestão flexível das reservas, garantindo eficiência e organização na operação da oficina (por exemplo: as reservas podem ser imediatamente guardadas por ordem cronológica, à medida que são acrescentadas).

Filas de espera para armazenar pré-reservas

As filas de espera (ou *queues*) são estruturas FIFO ("First-IN, First-OUT") que mantêm uma hierarquia entre os elementos que lhe são adicionados. Seguem um fluxo unidirecional, onde só é possível retirar o elemento inicial e adicionar novos elementos ao fim. Este traço garante um atendimento com base na ordem de chegada. Assim, quem chegar primeiro à fila de espera, terá prioridade caso uma reserva seja cancelada. A única exceção à regra será se existir uma correspondência cronológica exata com o cancelamento. É essencial tratar os pedidos na sequência em que são recebidos, para assegurar um atendimento justo e imparcial. As filas oferecem um mecanismo natural para implementar essa lógica, além de serem estruturas simples e intuitivas de manusear e de possuírem também capacidade de armazenamento dinâmico.

ESTRUTURAÇÃO DO CÓDIGO

03

De modo a simplificar o processo de implementação das diversas funções de gerenciamento, o código do projeto foi subdividido em cinco ficheiros de código fonte diferentes. Um desses ficheiros, o "source.h", é o responsável por criar as estruturas de dados necessárias ao correto funcionamento do programa e por armazenar os protótipos de todas as funções desenvolvidas nos ficheiros "source.c", "linkedLists.c" e "queues.c". Já o ficheiro "main.c" é o responsável por grande parte das interações com o utilizador.

"source.h"

Foram então criadas neste ficheiro as estruturas usadas para armazenar datas, horários, dados de clientes e as estruturas que definem as listas ligadas e as filas de espera. Estão também aqui os protótipos das funções relacionadas com a normalização da informação impressa, comparações de tempo, listas ligadas, filas de espera e com as funcionalidades a implementar.

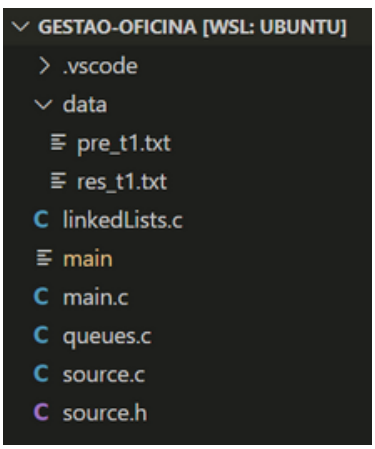


Fig. 1 - Divisão do projeto

```c
/* Struct de uma reserva ou pré-reserva */
typedef struct {
    char name[50];
    int service;
    Date date;
    Time time;
} Booking;
```

Fig. 2 - Principal *struct* de armazenamento de dados

ESTRUTURAÇÃO DO CÓDIGO

04

"source.c"

Neste ficheiro, foram desenvolvidas as seguintes funções: i) *checkName*; ii) *capitalizeWords*; iii) *printData*; iv) *dateCompare*; v) *timeCompability*; vi) *timeCompare*; vii) *timeFix*; viii) *reserve*; ix) *preReserve*; x) *cancelRes*; xi) *cancelPreRes*; xii) *listingRes*; xiii) *listingClient*; xiv) *updateServices*; xv) *checkFile*; xvi) *loadInfo* e xvii) *saveInfo*.

As primeiras três funções tratam da proteção na receção do nome de cliente (só permite letras) e da normalização da informação impressa (nome capitalizado, impressão bem formatada).

As funções iv), v) e vi) foram implementadas pela necessidade de comparar dois elementos, de modo a fazer uma inserção correta de um elemento nas reservas, tendo em conta a duração de cada serviço.

Já na *timeFix*, é efetuada uma correção do horário, necessária para a opção de menu que lista todas as vagas disponíveis, num dia à escolha.

As principais funções deste programa, a *reserve* e a *preReserve*, foram desenvolvidas tendo em conta a proteção contra a receção de informação inválida. Apenas se tudo estiver em conformidade (nome correto, serviço existente e data e hora superiores ou iguais à de sistema) é que se avança então para a chamada de outras funções auxiliares. No caso das reservas, é primeiro verificado se já existe alguma reserva igual nas listas. Em caso negativo, tenta inserir a nova reserva (pode haver uma incompatibilidade de horários com as que já lá estão). Se não for possível inserir, pergunta ao utilizador se quer mover os dados para uma fila de espera, em que fica em pré-reserva. Se a resposta for positiva, chama a função de pré-reserva, *preReserve*, que chama também uma função auxiliar de inserção na fila (e que mantém uma prioridade).

Nas função de cancelamento *cancelRes*, tenta-se remover um elemento da lista (através de nome, data e hora). Caso seja removido com sucesso, chama a função *toRes*, que irá tentar enquadrar a pré-reserva prioritária no lugar da reserva cancelada. Na *cancelPreRes*, tenta-se remover um item da fila.

ESTRUTURAÇÃO DO CÓDIGO

05

Na *listingRes*, tanto as reservas como as pré-reservas, são impressas das mais antigas para as mais recentes. Em oposição, na *listingClient* é utilizada a recursividade (nas reservas), para as imprimir pela ordem inversa.

De modo a facilitar a gestão dos serviços já realizados, foi feita uma função *UpdateServices*, que remove todos os elementos da lista e da fila com data e hora inferior ou igual à dada.

Finalmente, para a implementação dos ficheiros no programa, foram desenvolvidas as últimas três funções deste ficheiro. A finalidade da *checkFile* é meramente averiguar se o nome dado pelo utilizador tem os prefixos estabelecidos "pre_" ou "res_" (as reservas e pré-reservas são guardadas em ficheiros diferentes) e se tem o sufixo ".txt". Se tiver os prefixos, estes serão removidos. Se não tiver o sufixo, este é adicionado. As outras duas, *loadInfo* e *saveInfo*, servem apenas para carregar informação ou gravar o estado atual das reservas, num ficheiro à escolha.

"linkedLists.c"

Neste ficheiro, foram desenvolvidas as seguintes funções: i) *createList*; ii) emptyList; iii) destroyList; iv) *removeItemList*; v) *searchItem*; vi) searchClient; vii) printList; viii) insertItemOrder; ix) *printAvailableTime* e x) getListSize.

As funções i), ii) e iii) são semelhantes às funções do ficheiro que iremos analisar de seguida ("queues.c") e servem para criar a estrutura de dados, verificar se está vazia e destruí-la (libertar a memória que foi alocada).

Na *removeItemList*, pesquisa o elemento e tenta removê-lo. Para poder efetuar esta pesquisa, criou-se a *searchItem*.

Na *searchClient*, aplica-se a recursividade (mencionada anteriormente) para, quando o nome dado coincidir com o nome de cada uma das reservas, imprimir a informação inversamente (+ recentes para + antigas). Já a *printList*, é chamada quando se querem listar todas as reservas, e não apenas as de um cliente (+ antigas para + recentes).

A função *insertItemOrder* é também imprescindível para a inserção ordenada de elementos nas listas, evitando a sobreposição de horários e mantendo a ordem temporal das reservas.

ESTRUTURAÇÃO DO CÓDIGO

06

Como função extra ao projeto, foi ainda desenvolvida a *printAvailableTime*, que imprime todos os *slots* de reservas disponíveis em determinado dia, para que o funcionário da oficina consiga ter uma noção do nível de ocupação dos seus serviços.

Por fim, temos a *getListSize*, que apenas serve para armazenar o tamanho da lista, necessário a outras funções.

"queues.c"

Neste ficheiro, foram desenvolvidas as seguintes funções: i) *createQueue*; ii) *emptyQueue*; iii) *destroyQueue*; iv) *removeItemQueue*; v) *removePastItems*; vi) *addItem*; vii) *comparator*; viii) *printQueue* e ix) *toRes*.

As primeiras três foram já explicadas anteriormente, sendo a sua aplicação neste tipo de estruturas muito semelhante.

Quanto à função *removeItemQueue*, é importante salientar que foi necessário fazer uma adaptação ao processo natural de funcionamento das filas, tornando possível a remoção de qualquer elemento que a constitua. Isto porque, ao cancelar uma pré-reserva, tem que ser possível aceder a qualquer elemento da fila para o retirar.

A *removePastItems* foi implementada para, quando um serviço é realizado na oficina (através da *UpdateServices*) serem eliminadas todas as pré-reservas com data inferior à data de realização (note-se que as filas não são ordenadas cronologicamente, como acontece nas listas). Em contrapartida, existe a *addItem*, que lhe adiciona um novo elemento ao final.

Para listar as pré-reservas ordenadas por data e horário, foi necessário criar um *comparator*, usado na chamada do "qsort()", função que ordena um *array*, com base numa função de comparação. Esta chamada é feita na *printQueue*, que, conforme a opção escolhida, vai imprimir os dados pela ordem normal ou inversa (neste caso, não é utilizada a recursividade para a listagem específica).

Por fim, existe ainda a *toRes,* que, quando uma reserva é cancelada, tenta enviar uma das pré-reservas para as reservas, tendo em conta a prioridade em caso de correspondência total de data e hora, seguida da prioridade natural.

ESTRUTURAÇÃO DO CÓDIGO

07

"main.c"

Ao nível da "main.c", foram implementadas as funções que tratam as interações com o utilizador: i) *printMenu* e ii) *askData*, cujas funções são apenas imprimir o menu principal e pedir os dados necessários à marcação.

Para uma maior proteção do programa, foram também implementadas as funções *is_leap_year*, *is_valid_date*, *systemTime*, *checkDateValidity* e *checkTimeValidity*, que se interligam e fazem uma verificação exaustiva das datas e horas inseridas pelo utilizador, recorrendo à terceira para guardar a data de sistema e não permitir reservas anteriores à mesma.

Inicializadas as variáveis, estruturas e ponteiros necessários, entramos num ciclo "do - while", que irá imprimir o menu, no fim de cada operação realizada. O programa só é encerrado quando for selecionada a opção "0". Para o tratamento das nove escolhas possíveis, foi usado um "switch - case", que chama as devidas funções.

No fim de tudo, é fundamental libertar toda a memória que foi alocada ao longo do programa, através das funções *destroyList* e *destroyQueue*.

```c
/* Menu principal */
void printMenu() {
    printf("\e[1;32m\n[ 0 ]\e[0m Encerrar o programa\n");
    printf("\e[1;32m[ 1 ]\e[0m Carregar informação\n");
    printf("\e[1;32m[ 2 ]\e[0m Gravar o estado atual das reservas\n");
    printf("\e[1;32m[ 3 ]\e[0m Reservar lavagem ou manutenção\n");
    printf("\e[1;32m[ 4 ]\e[0m Cancelar reserva\n");
    printf("\e[1;32m[ 5 ]\e[0m Cancelar pré-reserva\n");
    printf("\e[1;32m[ 6 ]\e[0m Listar todas as reservas e pré-reservas\n");
    printf("\e[1;32m[ 7 ]\e[0m Listar as reservas e pré-reservas de um cliente\n");
    printf("\e[1;32m[ 8 ]\e[0m Realizar uma lavagem ou manutenção\n");
    printf("\e[1;32m[ 9 ]\e[0m Listar as reservas disponíveis por dia\n");
    printf("\n\e[1;32m>$ OPÇÃO: \e[0m");
}
```

Fig. 3 - Funcionalidades implementadas

08

Processo do desenvolvimento de *software*

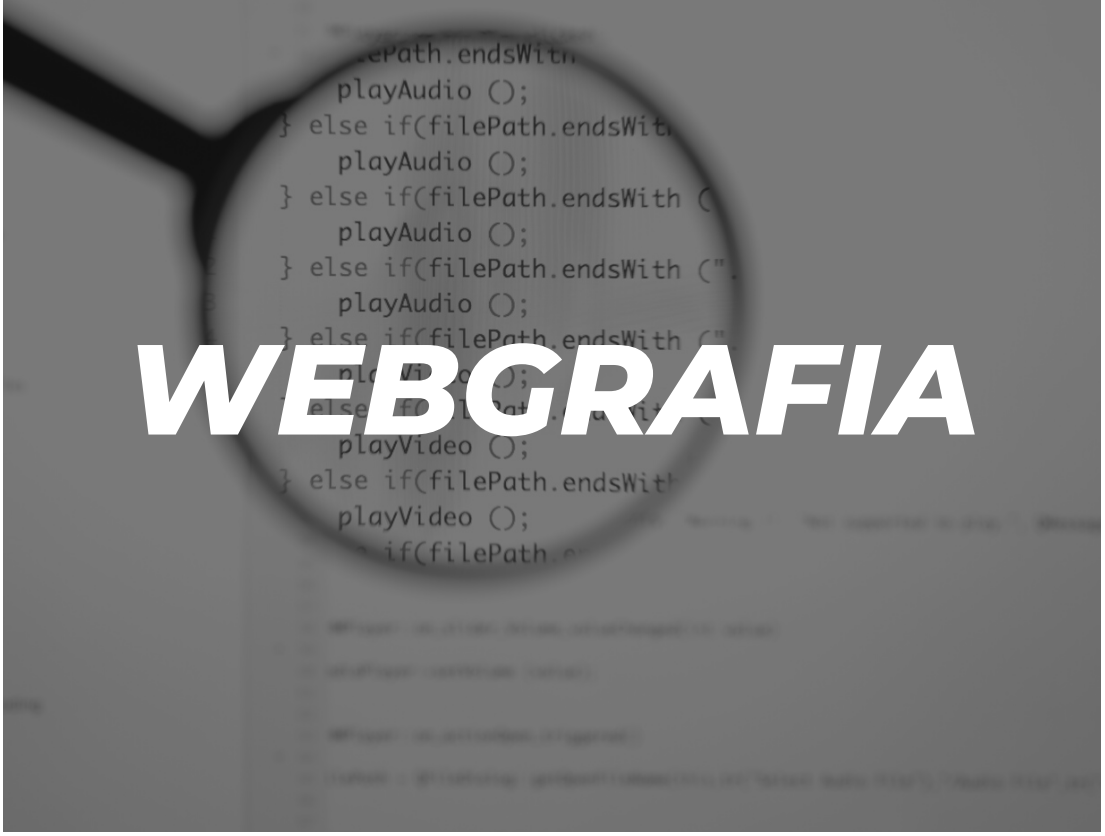
1. Análise das funcionalidades exigidas e esquematização do software;
2. *Design* da interface;
3. Construção das estruturas de dados necessárias;
4. Escolha e implementação dos métodos de armazenamento;
5. Criação dos protótipos das funções;
6. Desenvolvimento das funções prototipadas;
7. Testagem
8. Escrutínio do código e correção de bugs.

Fig. 4 - Processo típico de desenvolvimento

Precauções e boas práticas

Durante o processo de desenvolvimento, foram tomadas várias precauções, para garantir o melhor funcionamento do programa:

- Foram criadas medidas contra a introdução de informação errada por parte do utilizador;
- Foi tida em conta a forma de armazenamento (dinâmica *vs* não dinâmica), para não desperdiçar memória (com a devida libertação, no final);
- Com recurso a funções, foram evitadas repetições de código;
- Foram utilizados métodos de ordenação que recorrem a estruturas de dados auxiliares com apontadores para registos reais;
- Foi utilizada recursividade, de modo a garantir ganho de eficiência.



WEBGRAFIA

[1] DevDocs.io. *C Documentation*. [Online]. Disponível em: https://devdocs.io/c/. Acesso em: 15 de maio de 2023.

[2] Stack Overflow. *Get the current time in C*. [Online]. Disponível em: https://stackoverflow.com/questions/5141960/get-the-current-time-in-c. Acesso em: 30 de maio de 2023.

[3] Space-O Technologies. *Learn 6 Stages of Agile Software Development Process Steps. [Online]. Disponível em:* https://www.spaceotechnologies.com/blog/software-development-process/. Acesso em: 31 de maio de 2023.