

”

E-fólio A | Folha de resolução para E-fólio



UNIDADE CURRICULAR: Laboratório de Programação

CÓDIGO: 21178

DOCENTE: Nelson Russo

A preencher pelo estudante

NOME: Edgar Pinto Matuminy

N.º DE ESTUDANTE: 2500691

CURSO: Licenciatura Engenharia Informática

DATA DE ENTREGA: 18 de Abril de 2026

TRABALHO / RESOLUÇÃO:

1. Introdução

Este trabalho foi realizado com o objetivo principal de pegar num programa monolítico cheio de erros, que serve para gerir jardins comunitários, e transformá-lo num sistema funcional, organizado e modular.

O programa original, chamado GreenTrack, foi escrito por um programador iniciante e continha muitos problemas: desde erros de sintaxe (como nomes com espaços) até erros de lógica (como while (!feof(...))) e más práticas de programação (falta de free(), variáveis globais desnecessárias, etc.). Além disso, o código estava todo misturado, sem separação entre interface e implementação.

A minha abordagem foi dividida em duas grandes fases:

1. **Identificar e corrigir os erros**, analisei cada excerto fornecido no enunciado, cataloguei os problemas e apliquei as correções necessárias.
2. **Reorganizar o código de forma modular**, criei vários módulos (cada um com o seu .h e .c), usei static para esconder detalhes internos, e garanti que os dados ficam guardados em ficheiros CSV para não se perderem quando o programa termina.

Neste relatório vou explicar todos os erros que encontrei, mostrar como organizei o código, justificar as decisões que tomei e apresentar o código completo que desenvolvi.

2. Tarefa 1 – Erros Encontrados

Nesta tarefa, o enunciado fornecia vários excertos de código com erros. O meu trabalho foi encontrar **todos** os problemas, perceber a sua natureza e corrigi-los. No total, identifiquei **27 erros**, que organizei na tabela seguinte.

Tabela de erros identificados

#	Excerto	Linha	Erro	Categoria	Consequência	Correção
1	1	8	<code>while (!feof(f_plantas))</code>	● Lógica	A última linha era lida duas vezes	<code>while (fscanf(...) == 6)</code>
2	1	2	<code>MAX_PLANTAS</code> (espaço)	● Sintaxe	Não compila	<code>MAX_PLANTAS</code>
3	1	2	<code>MAX_REGAS</code> (espaço)	● Sintaxe	Não compila	<code>MAX_REGAS</code>
4	1	2	<code>MAX_TAREFAS</code> (espaço)	● Sintaxe	Não compila	<code>MAX_TAREFAS</code>
5	1	15	<code>data_plantio</code> (espaço)	● Sintaxe	Não compila	<code>data_plantio</code>
6	1	16	<code>ultima_rega</code> (espaço)	● Sintaxe	Não compila	<code>ultima_rega</code>
7	1	8	Falta verificação de limite do array	● Execução	Risco de estouro de memória	<code>Verificar total_plantas < MAX_PLANTAS</code>
8	2	4	<code>i <= total_plantas</code>	● Execução	Acede a posição inválida do array	<code>i < total_plantas</code>
9	3	3	<code>total_regas</code> (espaço)	● Sintaxe	Não compila	<code>total_regas</code>
10	3	3	<code>id_rega</code> (espaço)	● Sintaxe	Não compila	<code>id_rega</code>
11	3	4	<code>id_planta</code> (espaço)	● Sintaxe	Não compila	<code>id_planta</code>
12	3	6	<code>quantidade_agua</code> (espaço)	● Sintaxe	Não compila	<code>quantidade_agua</code>
13	3	2-7	Falta verificação de limite do array	● Execução	Estouro do array	<code>Verificar total_regas < MAX_REGAS</code>
14	3	2-7	Falta validar se a planta existe	● Lógica	Podia registrar rega para planta inexistente	<code>Usar obter_planta() para validar</code>
15	4	-	Função void sem retorno útil	● Design	Difícil reutilizar noutro contexto	<code>Passar a retornar int (nº de plantas)</code>
16	5	3	<code>total_tarefas</code> (espaço)	● Sintaxe	Não compila	<code>total_tarefas</code>
17	5	3	<code>id_tarefa</code> (espaço)	● Sintaxe	Não compila	<code>id_tarefa</code>
18	5	5	<code>data_prevista</code> (espaço)	● Sintaxe	Não compila	<code>data_prevista</code>
19	5	2-7	Falta verificação de limite do array	● Execução	Estouro do array	<code>Verificar total_tarefas < MAX_TAREFAS</code>
20	6	5	<code>if (tarefas[i].concluida = 0)</code>	● Lógica	Atribuição em vez de comparação	<code>if (tarefas[i].concluida == 0)</code>
21	7	4	<code>if (tarefas[i].id_tarefa = id)</code>	● Lógica	Atribuição em vez de comparação	<code>if (tarefas[i].id_tarefa == id)</code>
22	8	2-7	Só guardava as plantas	● Lógica	Regas e tarefas perdidas ao reiniciar	<code>Guardar também regas e tarefas</code>
23	8	4	Mensagem "Erro" genérica	● Má Prática	Não ajuda a perceber o problema	<code>Mensagem mais informativa</code>
24	9	5	<code>ultima_rega = 0</code>	● Lógica	Ficava sempre com data 01/01/2026	<code>Receber data_atual como parâmetro</code>
25	9	2-7	Falta verificação de limite do array	● Execução	Estouro do array	<code>Verificar antes de adicionar</code>
26	10	4	<code>scanf("%s", nome)</code>	● Execução	Não lê nomes com espaços	<code>Usar fgets()</code>
27	10	5-8	Faltavam leituras de id, data, intervalo	● Lógica	Dados incompletos	<code>Adicionar scanf para os campos em falta</code>

3. Tarefa 2 – Organização Modular

Depois de corrigir os erros, o passo seguinte foi **modularizar** o código. O objetivo era deixar de ter tudo num só ficheiro e passar a ter uma estrutura organizada, com módulos independentes e bem definidos.

3.1 Estrutura de diretórios que criei

projeto_greentrack/

include/ # Ficheiros .h (declarações)

src/ # Ficheiros .c (implementação)

data/ # Ficheiros CSV para guardar os dados

Makefile/ # Script de compilação

greentrack.exe # O programa já compilado

3.2 Módulos que criei

Estrutura Modular do Sistema		
Módulo	Ficheiros	O que faz
Gestor de Plantas	gestor_plantas.h e .c	Guarda a lista de plantas, permite adicionar, listar e verificar regas
Gestor de Regas	gestor_regas.h e .c	Guarda o histórico de regas, regista novas regas
Gestor de Tarefas	gestor_tarefas.h e .c	Cria tarefas, lista as pendentes, marca como concluídas
Gestor de Dados	gestor_dados.h e .c	Lê e escreve os ficheiros CSV (persistência)
Utils	utils.h e .c	Funções auxiliares (ler strings, validar datas, limpar buffer)
Main	main.c	Mostra o menu e chama as funções conforme a opção escolhida

3.3 O que é público e o que é privado

Para cada módulo, tomei a decisão de esconder os dados internos usando a palavra-chave static. Assim, ninguém consegue mexer diretamente nos arrays ou nos contadores, só podem usar as funções que eu disponibilizo no ficheiro .h.

		
Módulo: gestor_plantas		
Função / Dado	Visibilidade	Porquê
plantas[]	 Privada (static)	O array não deve ser acedido diretamente por outros módulos
total_plantas	 Privada (static)	O contador só pode ser alterado pelas funções do módulo
inicializar_plantas()	 Pública	É preciso limpar os dados no arranque
adicionar_planta()	 Pública	A interface principal para adicionar plantas
obter_planta()	 Pública	Permite aceder aos dados de forma controlada
listar_plantas()	 Pública	Mostra a lista no ecrã
verificar_necessidade_rega()	 Pública	Verifica quais plantas precisam de água
atualizar_ultima_rega()	 Pública	Atualiza a data da última rega
get_total_plantas()	 Pública	Dá acesso seguro ao contador
Módulo: gestor_regas		
Função / Dado	Visibilidade	Porquê
regas[]	 Privada (static)	Histórico de regas protegido
total_regas	 Privada (static)	Contador interno
inicializar_regas()	 Pública	Limpeza inicial
registrar_rega()	 Pública	Regista uma nova rega (com validação)
listar_regas_por_planta()	 Pública	Mostra histórico de uma planta
obter_rega()	 Pública	Acesso controlado aos dados
get_total_regas()	 Pública	Devolve quantas regas existem
Módulo: gestor_tarefas		
Função / Dado	Visibilidade	Porquê
tarefas[]	 Privada (static)	Lista de tarefas protegida
total_tarefas	 Privada (static)	Contador interno
inicializar_tarefas()	 Pública	Limpeza inicial
criar_tarefa()	 Pública	Cria uma nova tarefa
listar_tarefas_pendentes()	 Pública	Mostra as que ainda estão por fazer
concluir_tarefa()	 Pública	Marca uma tarefa como concluída
obter_tarefa()	 Pública	Acesso controlado aos dados
get_total_tarefas()	 Pública	Devolve quantas tarefas existem

3.4 Como garanti o encapsulamento

O encapsulamento foi conseguido de três formas:

1. **static nas variáveis globais dentro dos .c:** Por exemplo, static Planta plantas[MAX_PLANTAS]. Isto faz com que o array só seja visível dentro do próprio ficheiro. Nenhum outro módulo consegue mexer diretamente nele.
2. **Separação entre .h e .c:** Nos ficheiros .h só coloquei aquilo que é mesmo preciso para usar o módulo: protótipos de funções, constantes e a definição das estruturas (quando são públicas). A implementação verdadeira ficou escondida nos .c.
3. **Funções de acesso:** Em vez de permitir o acesso direto a variáveis como total_plantas, criei funções como get_total_plantas() que devolvem o valor sem permitir que seja alterado por fora.

3.4 Como garanti o encapsulamento

O encapsulamento foi conseguido de três formas:

1. **static nas variáveis globais dentro dos .c:** Por exemplo, `static Planta plantas[MAX_PLANTAS]`. Isto faz com que o array só seja visível dentro do próprio ficheiro. Nenhum outro módulo consegue mexer diretamente nele.
2. **Separação entre .h e .c:** Nos ficheiros .h só coloquei aquilo que é mesmo preciso para usar o módulo: protótipos de funções, constantes e a definição das estruturas (quando são públicas). A implementação verdadeira ficou escondida nos .c.
3. **Funções de acesso:** Em vez de permitir o acesso direto a variáveis como `total_plantas`, criei funções como `get_total_plantas()` que devolvem o valor sem permitir que seja alterado por fora.

3.5 Porque é que organizei assim?

A minha ideia foi separar o código por **responsabilidades** (o famoso princípio da responsabilidade única). Cada módulo trata de um aspeto específico do sistema:

- **Plantas** – tudo o que tem a ver com plantas (adicionar, listar, verificar regas)
- **Regas** – histórico de regas
- **Tarefas** – criação e gestão de tarefas
- **Dados** – guardar e carregar de ficheiros
- **Utils** – pequenas ajudas que são úteis em vários sítios

Alternativas que pensei e rejeitei:

- **Juntar tudo num único módulo** – era o que estava no código original, mas assim perdia-se a organização e ficava difícil dar manutenção.
- **Usar tipos opacos** – pensei nisso, mas achei que ia complicar desnecessariamente o projeto, que é de média dimensão.
- **Separar o gestor_dados em três módulos (um para cada tipo de dado)** – ia repetir muito código (abrir/fechar ficheiros, etc.), por isso preferi manter tudo num só módulo.

3.6 Vantagens da organização que escolhi

- **Fica mais fácil dar manutenção** – se precisar de mudar alguma coisa nas regas, só mexo no gestor_regas.c, o resto do programa continua a funcionar.
- **O código fica mais limpo** – o main.c ficou pequeno e só chama funções, não tem lógica complicada.
- **Dá para reutilizar** – se um dia precisar de usar a gestão de tarefas noutra projeto, é só copiar os ficheiros gestor_tarefas.h e .c.
- **Testar é mais fácil** – posso testar cada módulo separadamente.

3.7 Dependências entre módulos

As dependências que criei são:

- gestor_regas precisa do gestor_plantas (para validar se a planta existe antes de registar uma rega)
- gestor_dados precisa de todos os outros (porque guarda e carrega plantas, regas e tarefas)
- main precisa de todos para orquestrar o programa

Uma coisa importante: **não há dependências circulares**. A seta vai sempre numa direção, o que torna o código mais simples de perceber e de compilar.

4. Funcionamento geral do programa

Para comprovar que o programa foi desenvolvido corretamente e que todas as funcionalidades estão operacionais, realizei uma bateria de testes práticos. Esta secção documenta os testes efetuados e apresenta evidências visuais do funcionamento do GreenTrack.

4.1 Teste de introdução de dados

Passo 1 – Adicionar uma planta

```
edgarpintomatuminy@VST1164885111:/mnt/c/Users/edgar.pinto/Documents/E-folio A - Laboratorio de Programacao/E-folio A/projeto_greentack$ gcc -Wall -Wextra -Wpedantic -I./include src/*.c -o greentack.exe
edgarpintomatuminy@VST1164885111:/mnt/c/Users/edgar.pinto/Documents/E-folio A - Laboratorio de Programacao/E-folio A/projeto_greentack$ ./greentack.exe

===== Menu GREENTACK =====
1. Listar plantas
2. Adicionar planta
3. Registrar rega
4. Verificar necessidades de rega
5. Listar tarefas pendentes
6. Criar tarefa
7. Concluir tarefa
8. Guardar e sair

===== INTRODUIZ DADOS ABAIXO =====

Opcao: 2
ID da planta: 1
Nome: Rosa
Especie: Rosa sp
Data de plantio (DD/MM/AAAA): 18/04/2026
Intervalo de rega (dias): 3
Dados guardados com sucesso!
```

Passo 2 – Listar a planta

```
===== Menu GREENTRACK =====
1. Listar plantas
2. Adicionar planta
3. Registrar rega
4. Verificar necessidades de rega
5. Listar tarefas pendentes
6. Criar tarefa
7. Concluir tarefa
8. Guardar e sair

===== INTRODUIZIR DADOS ABAIXO =====

Opcao: 1

===== LISTA DE PLANTAS =====
ID: 1 | Nome: Rosa | Espécie: Rosa sp | Plantio: 18/04/2026 | Rega: 3 dias
```

Passo 3 – Registrar uma rega

```
===== INTRODUIZIR DADOS ABAIXO =====

Opcao: 3
ID da planta: 1
Quantidade de água (ml): 500
Rega registada com sucesso!
Dados guardados com sucesso!
```

Passo 4 – Criar uma tarefa

```
===== INTRODUIZIR DADOS ABAIXO =====

Opcao: 6
Descrição da tarefa: Regar roseiras
Data prevista (timestamp): 15
Tarefa criada com sucesso!
Dados guardados com sucesso!
```

Passo 5 – Listar tarefas pendentes

```
===== INTRODUIZIR DADOS ABAIXO =====

Opcao: 5

===== TAREFAS PENDENTES =====
ID: 1 | Regar roseiras (prevista: 15)
```

Passo 6 – Concluir a tarefa

```
===== INTRODUIZIR DADOS ABAIXO =====

Opcao: 7
ID da tarefa: 1
Tarefa "Regar roseiras" concluída com sucesso!
Dados guardados com sucesso!
```

Passo 7 – Guardar e sair

```
===== INTRODUIZIR DADOS ABAIXO =====

Opcao: 8
Dados guardados com sucesso!
Dados guardados. A sair...
edgarpintonat@uminy@VST1164805111:/mnt/c/Users/edgar.pinto/Documents/E-fólio A - Laboratorio de Programação/E-fólio A/projeto_greentrack$
```

5. Conclusão

5.1 Desafios que enfrentei

O maior desafio foi, sem dúvida, **identificar todos os erros** nos excertos. Alguns eram óbvios (como nomes com espaços), mas outros eram mais subtis, como o `while (!feof(...))` que causa a leitura repetida da última linha, ou a atribuição `=` em vez de comparação `==`. Foi preciso testar mentalmente cada pedaço de código para perceber o que realmente estava mal.

Outro desafio foi **organizar a modularização** sem deixar dependências estranhas. Tive de pensar bem em que funções deviam ser públicas e quais deviam ficar privadas com `static`.

5.2 O que aprendi com este trabalho

- **A importância dos header guards:** sem eles, incluir o mesmo ficheiro duas vezes dá erros chatinhos de resolver.
- **Como static ajuda a esconder informação:** os arrays e contadores ficam protegidos, e só se acede a eles através de funções controladas.
- **A separação entre .h e .c não é só uma formalidade:** é uma forma de organizar o pensamento e o código.
- **A persistência em CSV é simples mas eficaz:** dá para guardar os dados e recuperá-los na próxima execução sem grandes complicações.

5.3 O que podia melhorar no futuro

- **Usar alocação dinâmica:** em vez de ter um limite fixo de 100 plantas, 500 regas e 200 tarefas, podia usar `malloc` e `realloc` para expandir conforme necessário.
- **Melhorar a gestão das datas:** a função `obter_data_atual()` está simplificada (devolve sempre 100). Numa versão mais completa, podia usar a biblioteca `time.h` para obter a data real do sistema.
- **Adicionar mais validações:** por exemplo, não permitir duas plantas com o mesmo nome, ou verificar se a data de plantio é anterior à data atual.

6. Código Desenvolvido

Aqui apresento o código completo de todos os ficheiros que criei. Todo o código está comentado, com explicações do que cada parte faz.

6.1 include/gestor_plantas.h

```
/**
 * @file gestor_plantas.h
 * @brief Interface do módulo de gestão de plantas.
 *
 * Este ficheiro contém os protótipos das funções e as estruturas
 * necessárias para gerir a lista de plantas do jardim.
 * Fui eu, Edgar Pinto Matuminy, que escrevi este módulo com base
 * nos excertos fornecidos pela UC.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

#ifndef GESTOR_PLANTAS_H
#define GESTOR_PLANTAS_H

/* ===== CONSTANTES ===== */

/** Número máximo de plantas que o sistema pode guardar (escolhi 100) */
#define MAX_PLANTAS 100

/** Tamanho máximo para o nome de uma planta (50 caracteres chega bem) */
#define MAX_NOME 50

/** Tamanho máximo para o nome da espécie (também 50) */
#define MAX_ESPECIE 50

/** Tamanho máximo para uma data no formato DD/MM/AAAA (10 letras) */
#define MAX_DATA 11

/* ===== ESTRUTURA ===== */

/**
 * @struct Planta
 * @brief Representa uma planta no jardim.
 *
 * Guarda toda a informação importante sobre uma planta:
 * - identificador único, nome, espécie, data de plantio,
 * - intervalo de rega recomendado e a data da última rega.
 */
typedef struct {
    int id; /*< Número único da planta */
    char nome[MAX_NOME]; /*< Nome que lhe quisermos dar */
    char especie[MAX_ESPECIE]; /*< Espécie científica ou popular */
    char data_plantio[MAX_DATA]; /*< Quando foi plantada (DD/MM/AAAA) */
    int intervalo_rega; /*< De quantos em quantos dias precisa de água */
    int ultima_rega; /*< Dia da última rega (timestamp) */
} Planta;

/* ===== FUNÇÕES ===== */

/**
 * @brief Prepara o array de plantas, limpando tudo o que lá estiver.
 * @return 1 se tudo correu bem.
 */
int inicializar_plantas(void);
```

```

/**
 * @brief Adiciona uma planta nova ao sistema.
 * @param id O número único da planta.
 * @param nome O nome que lhe queremos dar.
 * @param especie A espécie da planta.
 * @param data_plantio Quando foi plantada (DD/MM/AAAA).
 * @param intervalo De quantos em quantos dias deve ser regada.
 * @param data_atual A data de hoje (timestamp) para marcar a última rega.
 * @return 1 se conseguiu adicionar, 0 se houve erro (limite excedido ou ID repetido).
 */
int adicionar_planta(int id, const char* nome, const char* especie,
                    const char* data_plantio, int intervalo, int data_atual);

/**
 * @brief Procura uma planta pelo seu ID e copia os dados para destino.
 * @param id O número da planta que queremos encontrar.
 * @param destino Ponteiro onde vamos guardar os dados da planta.
 * @return 1 se encontrou a planta, 0 se não existe nenhuma com esse ID.
 */
int obter_planta(int id, Planta* destino);

/**
 * @brief Mostra no ecrã uma lista bonita com todas as plantas.
 * @return O número de plantas que mostrámos (0 se não houver nenhuma).
 */
int listar_plantas(void);

/**
 * @brief Verifica quais plantas já estão a precisar de água.
 * @param data_atual A data de hoje (timestamp).
 * @return Quantas plantas precisam de ser regadas.
 */
int verificar_necessidade_rega(int data_atual);

/**
 * @brief Actualiza a data da última rega de uma planta.
 * @param id_planta Qual é a planta que acabámos de regar.
 * @param data_rega A data em que a rega aconteceu (timestamp).
 * @return 1 se conseguiu actualizar, 0 se a planta não existe.
 */
int atualizar_ultima_rega(int id_planta, int data_rega);

/**
 * @brief Diz quantas plantas estão registadas neste momento.
 * @return O número total de plantas.
 */
int get_total_plantas(void);

#endif /* GESTOR_PLANTAS_H */

```

6.2 src/gestor_plantas.c

```

/**
 * @file gestor_plantas.c
 * @brief Implementação do módulo de gestão de plantas.
 *
 * Aqui é onde está a verdadeira lógica para guardar, listar e actualizar
 * as plantas. As variáveis principais (plantas[] e total_plantas) são
 * 'static' para que ninguém de fora lhes mexa directamente.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

```

```

#include <stdio.h>
#include <string.h>
#include "gestor_plantas.h"

/* ===== DADOS PRIVADOS ===== */

/**
 * @brief Array onde guardo todas as plantas.
 * Está 'static' para ficar escondido dentro deste ficheiro.
 */
static Planta plantas[MAX_PLANTAS];

/**
 * @brief Quantas plantas já foram adicionadas.
 * Também é 'static' para que só este módulo mexa neste contador.
 */
static int total_plantas = 0;

/* ===== IMPLEMENTAÇÃO ===== */

int inicializar_plantas(void) {
    /* Coloco o contador a zero e apago toda a memória do array */
    total_plantas = 0;
    memset(plantas, 0, sizeof(plantas));
    return 1;
}

int adicionar_planta(int id, const char* nome, const char* especie,
                    const char* data_plantio, int intervalo, int data_atual) {

    /* 1.º passo: verificar se ainda há espaço */
    if (total_plantas >= MAX_PLANTAS) {
        printf("Erro: limite máximo de plantas atingido (%d)\n", MAX_PLANTAS);
        return 0;
    }

    /* 2.º passo: garantir que este ID ainda não foi usado */
    for (int i = 0; i < total_plantas; i++) {
        if (plantas[i].id == id) {
            printf("Erro: já existe uma planta com o ID %d\n", id);
            return 0;
        }
    }

    /* 3.º passo: copiar os dados para o próximo lugar vazio */
    plantas[total_plantas].id = id;
    strcpy(plantas[total_plantas].nome, nome);
    strcpy(plantas[total_plantas].especie, especie);
    strcpy(plantas[total_plantas].data_plantio, data_plantio);
    plantas[total_plantas].intervalo_rega = intervalo;
    plantas[total_plantas].ultima_rega = data_atual;

    /* 4.º passo: aumentar o contador */
    total_plantas++;

    return 1;
}

int obter_planta(int id, Planta* destino) {
    /* Vou vasculhar uma a uma até encontrar a planta com o ID desejado */
    for (int i = 0; i < total_plantas; i++) {
        if (plantas[i].id == id) {
            *destino = plantas[i]; /* copio os dados para o destino */
            return 1;
        }
    }
    return 0; /* não encontrei nenhuma planta com este ID */
}

```

```

}

int listar_plantas(void) {
    /* Se não há plantas, aviso logo e saio */
    if (total_plantas == 0) {
        printf("Nenhuma planta registrada.\n");
        return 0;
    }

    /* Mostro um cabeçalho bonito e depois cada planta */
    printf("\n===== LISTA DE PLANTAS =====\n");
    for (int i = 0; i < total_plantas; i++) {
        printf("ID: %d | Nome: %s | Espécie: %s | Plantio: %s | Rega: %d dias\n",
            plantas[i].id, plantas[i].nome, plantas[i].especie,
            plantas[i].data_plantio, plantas[i].intervalo_rega);
    }
    return total_plantas;
}

int verificar_necessidade_rega(int data_atual) {
    int count = 0; /* vou contar quantas precisam de água */

    printf("\n===== PLANTAS QUE PRECISAM DE REGA =====\n");

    for (int i = 0; i < total_plantas; i++) {
        /* quantos dias se passaram desde a última rega */
        int dias = data_atual - plantas[i].ultima_rega;

        /* se passou mais tempo do que o intervalo, está na hora */
        if (dias >= plantas[i].intervalo_rega) {
            printf("Planta \"%s\" (ID: %d) precisa de rega! (%d dias sem rega)\n",
                plantas[i].nome, plantas[i].id, dias);
            count++;
        }
    }

    if (count == 0) {
        printf("Nenhuma planta precisa de rega no momento.\n");
    }

    return count;
}

/**
 * @brief Atualiza a data da última rega de uma planta
 *
 * @param id_planta ID da planta regada
 * @param data_rega Data da rega (timestamp)
 * @return 1 se sucesso, 0 se planta não encontrada
 */
int atualizar_ultima_rega(int id_planta, int data_rega) {
    for (int i = 0; i < total_plantas; i++) {
        if (plantas[i].id == id_planta) {
            plantas[i].ultima_rega = data_rega;
            return 1;
        }
    }
    return 0; /* planta não encontrada */
}

/**
 * @brief Retorna o número total de plantas
 *
 * Função getter para aceder ao contador privado.
 *
 * @return Número total de plantas
 */
int get_total_plantas(void) {

```

```
    return total_plantas;
}
```

6.3 include/gestor_regas.h

```
/**
 * @file gestor_regas.h
 * @brief Interface do módulo de regas.
 *
 * Declarações para guardar o histórico de regas e funções
 * para registar novas regas e consultar regas antigas.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

#ifndef GESTOR_REGAS_H
#define GESTOR_REGAS_H

#include "gestor_plantas.h"

/** Número máximo de regas que consigo guardar (500 parece razoável) */
#define MAX_REGAS 500

/**
 * @struct Rega
 * @brief Guarda a informação de uma rega.
 */
typedef struct {
    int id_rega;      /**< Identificador único da rega */
    int id_planta;    /**< A planta que foi regada */
    int data_rega;    /**< Em que dia aconteceu a rega (timestamp) */
    int quantidade_agua; /**< Quantos ml de água foram usados */
} Rega;

/** ===== FUNÇÕES ===== */

/**
 * @brief Prepara o array de regas, limpando os dados antigos.
 * @return 1 se tudo correu bem.
 */
int inicializar_regas(void);

/**
 * @brief Regista uma nova rega.
 * @param id_planta Qual é a planta que foi regada.
 * @param data Quando foi regada (timestamp).
 * @param quantidade Quantos ml de água.
 * @return 1 se conseguiu registar, 0 se houve erro (limite excedido ou planta inválida).
 */
int registrar_rega(int id_planta, int data, int quantidade);

/**
 * @brief Mostra todas as regas que foram feitas a uma determinada planta.
 * @param id_planta A planta que nos interessa.
 * @return Quantas regas encontrámos.
 */
int listar_regas_por_planta(int id_planta);

/**
 * @brief Procura uma rega pelo seu ID e devolve os dados.
 * @param id O número da rega.
 * @param destino Onde guardar os dados.
 * @return 1 se encontrou, 0 se não existe.
 */
```

```

int obter_rega(int id, Rega* destino);

/**
 * @brief Quantas regas estão guardadas no total.
 * @return O número total de regas.
 */
int get_total_regas(void);

#endif /* GESTOR_REGAS_H */

```

6.4 src/gestor_regas.c

```

/**
 * @file gestor_regas.c
 * @brief Implementação do módulo de regas.
 *
 * Aqui é onde guardo mesmo as regas (num array 'static')
 * e implemento a lógica para registar novas regas e
 * mostrar o histórico.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

#include <stdio.h>
#include <string.h>
#include "gestor_regas.h"

/* ===== DADOS PRIVADOS ===== */

/** Array onde fica guardado o histórico de todas as regas */
static Rega regas[MAX_REGAS];

/** Quantas regas já foram registadas até agora */
static int total_regas = 0;

/* ===== IMPLEMENTAÇÃO ===== */

int inicializar_regas(void) {
    total_regas = 0;
    memset(regas, 0, sizeof(regas));
    return 1;
}

int registrar_rega(int id_planta, int data, int quantidade) {
    /* 1.º: verificar se ainda cabe mais uma rega */
    if (total_regas >= MAX_REGAS) {
        printf("Erro: limite máximo de regas atingido (%d)\n", MAX_REGAS);
        return 0;
    }

    /* 2.º: confirmar que a planta existe mesmo */
    Planta p;
    if (!obter_planta(id_planta, &p)) {
        printf("Erro: planta com ID %d não encontrada\n", id_planta);
        return 0;
    }

    /* 3.º: guardar a nova rega */
    regas[total_regas].id_rega = total_regas + 1; /* ID automático */
    regas[total_regas].id_planta = id_planta;
    regas[total_regas].data_rega = data;
    regas[total_regas].quantidade_agua = quantidade;
    total_regas++;

    /* 4.º: actualizar a última rega na ficha da planta */
}

```

```

    atualizar_ultima_rega(id_planta, data);

    printf("Rega registada com sucesso!\n");
    return 1;
}

int listar_regas_por_planta(int id_planta) {
    int count = 0;

    printf("\n===== HISTÓRICO DE REGAS (Planta ID: %d) =====\n", id_planta);

    for (int i = 0; i < total_regas; i++) {
        if (regas[i].id_planta == id_planta) {
            printf("Data: %d | Água: %d ml\n", regas[i].data_rega, regas[i].quantidade_agua);
            count++;
        }
    }

    if (count == 0) {
        printf("Nenhuma rega registada para esta planta.\n");
    }

    return count;
}

int obter_rega(int id, Rega* destino) {
    for (int i = 0; i < total_regas; i++) {
        if (regas[i].id_rega == id) {
            *destino = regas[i];
            return 1;
        }
    }
    return 0;
}

int get_total_regas(void) {
    return total_regas;
}

```

6.5 include/gestor_tarefas.h

```

/**
 * @file gestor_tarefas.h
 * @brief Interface do módulo de tarefas.
 *
 * Declarações para criar tarefas, listar as que ainda estão
 * pendentes e marcar tarefas como concluídas.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

#ifndef GESTOR_TAREFAS_H
#define GESTOR_TAREFAS_H

/** Número máximo de tarefas que o sistema aguenta (200 está bom) */
#define MAX_TAREFAS 200

/** Tamanho máximo da descrição de uma tarefa */
#define MAX_DESCRICAO 100

/**
 * @struct Tarefa
 * @brief Guarda a informação de uma tarefa.
 */

```

```

typedef struct {
    int id_tarefa;           /**< Número único da tarefa */
    char descricao[MAX_DESCRICAO]; /**< O que é preciso fazer */
    int data_prevista;       /**< Até quando deve estar concluída (timestamp) */
    int concluida;          /**< 0 = ainda não feita, 1 = já está concluída */
} Tarefa;

/* ===== FUNÇÕES ===== */

/**
 * @brief Prepara o array de tarefas, limpando dados antigos.
 * @return 1 se tudo correu bem.
 */
int inicializar_tarefas(void);

/**
 * @brief Cria uma nova tarefa.
 * @param descricao O que é preciso fazer.
 * @param data_prevista Data limite para concluir (timestamp).
 * @return 1 se conseguiu criar, 0 se houve erro (limite excedido).
 */
int criar_tarefa(const char* descricao, int data_prevista);

/**
 * @brief Mostra todas as tarefas que ainda estão por fazer.
 * @return Quantas tarefas pendentes foram mostradas.
 */
int listar_tarefas_pendentes(void);

/**
 * @brief Marca uma tarefa como concluída.
 * @param id O número da tarefa que foi concluída.
 * @return 1 se conseguiu marcar, 0 se a tarefa não existe.
 */
int concluir_tarefa(int id);

/**
 * @brief Procura uma tarefa pelo seu ID.
 * @param id Número da tarefa.
 * @param destino Onde guardar os dados.
 * @return 1 se encontrou, 0 se não existe.
 */
int obter_tarefa(int id, Tarefa* destino);

/**
 * @brief Quantas tarefas estão registradas no total.
 * @return O número total de tarefas.
 */
int get_total_tarefas(void);

#endif /* GESTOR_TAREFAS_H */

```

6.6 src/gestor_tarefas.c

```

/**
 * @file gestor_tarefas.c
 * @brief Implementação do módulo de tarefas.
 *
 * Criação, listagem e conclusão de tarefas. Uso um array
 * 'static' para guardar tudo e funções 'get_total_tarefas'
 * para dar acesso seguro ao contador.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026

```

```

*/

#include <stdio.h>
#include <string.h>
#include "gestor_tarefas.h"

/* ===== DADOS PRIVADOS ===== */

/** Array onde guardo todas as tarefas */
static Tarefa tarefas[MAX_TAREFAS];

/** Quantas tarefas já foram criadas */
static int total_tarefas = 0;

/* ===== IMPLEMENTAÇÃO ===== */

int inicializar_tarefas(void) {
    total_tarefas = 0;
    memset(tarefas, 0, sizeof(tarefas));
    return 1;
}

int criar_tarefa(const char* descricao, int data_prevista) {
    /* Ver se ainda cabe mais uma tarefa */
    if (total_tarefas >= MAX_TAREFAS) {
        printf("Erro: limite máximo de tarefas atingido (%d)\n", MAX_TAREFAS);
        return 0;
    }

    /* Preencher os dados da nova tarefa */
    tarefas[total_tarefas].id_tarefa = total_tarefas + 1;
    strcpy(tarefas[total_tarefas].descricao, descricao);
    tarefas[total_tarefas].data_prevista = data_prevista;
    tarefas[total_tarefas].concluida = 0; /* começa como pendente */
    total_tarefas++;

    printf("Tarefa criada com sucesso!\n");
    return 1;
}

int listar_tarefas_pendentes(void) {
    int count = 0;

    printf("\n===== TAREFAS PENDENTES =====\n");

    for (int i = 0; i < total_tarefas; i++) {
        if (tarefas[i].concluida == 0) { /* só as que ainda não foram feitas */
            printf("ID: %d | %s (prevista: %d)\n",
                tarefas[i].id_tarefa, tarefas[i].descricao, tarefas[i].data_prevista);
            count++;
        }
    }

    if (count == 0) {
        printf("Nenhuma tarefa pendente.\n");
    }

    return count;
}

int concluir_tarefa(int id) {
    for (int i = 0; i < total_tarefas; i++) {
        if (tarefas[i].id_tarefa == id) {
            if (tarefas[i].concluida == 1) {
                printf("Tarefa já estava concluída.\n");
                return 0;
            }
        }
    }
}

```

```

        tarefas[i].concluida = 1;
        printf("Tarefa \"%s\" concluída com sucesso!\n", tarefas[i].descricao);
        return 1;
    }
}

printf("Erro: tarefa com ID %d não encontrada\n", id);
return 0;
}

/**
 * @brief Obtém dados de uma rega pelo ID
 *
 * @param id ID da rega
 * @param destino Ponteiro onde copiar os dados
 * @return 1 se encontrada, 0 se não encontrada
 */
int obter_tarefa(int id, Tarefa* destino) {
    for (int i = 0; i < total_tarefas; i++) {
        if (tarefas[i].id_tarefa == id) {
            *destino = tarefas[i];
            return 1;
        }
    }
    return 0;
}

/**
 * @brief Retorna o número total de regas
 *
 * Função getter para aceder ao contador privado.
 *
 * @return Número total de regas
 */
int get_total_tarefas(void) {
    return total_tarefas;
}

```

6.7 include/gestor_dados.h

```

/**
 * @file gestor_dados.h
 * @brief Interface do módulo de persistência (guardar / carregar).
 *
 * Declarações para carregar os dados dos ficheiros CSV
 * e para os guardar outra vez quando saímos.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

#ifndef GESTOR_DADOS_H
#define GESTOR_DADOS_H

/**
 * @brief Carrega tudo o que estava guardado nos ficheiros CSV.
 * @return 1 se conseguiu carregar (mesmo que alguns ficheiros não existam).
 */
int carregar_dados(void);

/**
 * @brief Guarda tudo nos ficheiros CSV.
 * @return 1 se conseguiu guardar, 0 se houve erro ao escrever.
 */
int guardar_dados(void);

#endif /* GESTOR_DADOS_H */

```

6.8 src/gestor_dados.c

```
/**
 * @file gestor_dados.c
 * @brief Implementação da persistência em CSV.
 *
 * Este módulo é o responsável por ler os ficheiros plantas.csv,
 * regas.csv e tarefas.csv quando o programa arranca, e por
 * escrever neles sempre que fazemos uma alteração.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gestor_dados.h"
#include "gestor_plantas.h"
#include "gestor_regas.h"
#include "gestor_tarefas.h"

/* Caminhos dos ficheiros onde guardo os dados */
#define FICHEIRO_PLANTAS "data/plantas.csv"
#define FICHEIRO_REGAS "data/regas.csv"
#define FICHEIRO_TAREFAS "data/tarefas.csv"
/**
 * @brief Carrega os dados dos ficheiros CSV
 *
 * Tenta abrir cada ficheiro CSV. Se existir, lê os dados
 * linha por linha e adiciona aos arrays internos.
 * Se não existir, apenas continua (cria ficheiros novos mais tarde).
 *
 * @return 1 se sucesso
 */
int carregar_dados(void) {
    FILE* f;
    int id, intervalo, ultima_rega, id_planta, data_rega, quantidade;
    int data_prevista, concluida;
    char nome[MAX_NOME], especie[MAX_ESPECIE], data_plantio[MAX_DATA];
    char descricao[MAX_DESCRICAO];
    int linha_lida;

    /* ----- PLANTAS ----- */
    f = fopen(FICHEIRO_PLANTAS, "r");
    if (f != NULL) {
        inicializar_plantas();
        while (1) {
            linha_lida = fscanf(f, "%d,%49[^\n],%49[^\n],%10[^\n],%d,%d\n",
                                &id, nome, especie, data_plantio, &intervalo, &ultima_rega);
            if (linha_lida != 6) break; /* saio quando acabar o ficheiro */
            adicionar_planta(id, nome, especie, data_plantio, intervalo, ultima_rega);
        }
        fclose(f);
        printf("Plantas carregadas: %d\n", get_total_plantas());
    }

    /* ----- REGAS ----- */
    f = fopen(FICHEIRO_REGAS, "r");
    if (f != NULL) {
        inicializar_regas();
        while (1) {
            linha_lida = fscanf(f, "%d,%d,%d,%d\n", &id, &id_planta, &data_rega, &quantidade);
            if (linha_lida != 4) break;
            registrar_rega(id_planta, data_rega, quantidade);
        }
    }
}
```

```

    fclose(f);
    printf("Regas carregadas: %d\n", get_total_regas());
}

/* ----- TAREFAS ----- */
f = fopen(FICHEIRO_TAREFAS, "r");
if (f != NULL) {
    inicializar_tarefas();
    while (1) {
        linha_lida = fscanf(f, "%d,%99[^\n],%d,%d\n", &id, descricao, &data_prevista, &concluida);
        if (linha_lida != 4) break;
        criar_tarefa(descricao, data_prevista);
        if (concluida == 1) {
            concluir_tarefa(id);
        }
    }
    fclose(f);
    printf("Tarefas carregadas: %d\n", get_total_tarefas());
}

return 1;
}

int guardar_dados(void) {
    FILE* f;
    Planta p;
    Rega r;
    Tarefa t;

    /* ----- GUARDAR PLANTAS ----- */
    f = fopen(FICHEIRO_PLANTAS, "w");
    if (f == NULL) {
        printf("Erro ao abrir %s para escrita\n", FICHEIRO_PLANTAS);
        return 0;
    }
    /* Escreve todas as plantas linha por linha */
    for (int i = 0; i < get_total_plantas(); i++) {
        if (obter_planta(i + 1, &p)) {
            fprintf(f, "%d,%s,%s,%s,%d,%d\n",
                p.id, p.nome, p.especie, p.data_plantio,
                p.intervalo_rega, p.ultima_rega);
        }
    }
    fclose(f);

    /* ----- GUARDAR REGAS ----- */
    f = fopen(FICHEIRO_REGAS, "w");
    if (f == NULL) {
        printf("Erro ao abrir %s para escrita\n", FICHEIRO_REGAS);
        return 0;
    }
    /* Escreve todas as regas linha por linha */
    for (int i = 0; i < get_total_regas(); i++) {
        if (obter_rega(i + 1, &r)) {
            fprintf(f, "%d,%d,%d,%d\n", r.id_rega, r.id_planta, r.data_rega, r.quantidade_agua);
        }
    }
    fclose(f);

    /* ----- GUARDAR TAREFAS ----- */
    f = fopen(FICHEIRO_TAREFAS, "w");
    if (f == NULL) {
        printf("Erro ao abrir %s para escrita\n", FICHEIRO_TAREFAS);
        return 0;
    }
    /* Escreve todas as tarefas linha por linha */
    for (int i = 0; i < get_total_tarefas(); i++) {

```

```

        if (obter_tarefa(i + 1, &t)) {
            fprintf(f, "%d,%s,%d,%d\n", t.id_tarefa, t.descricao, t.data_prevista, t.concluida);
        }
    }
    fclose(f);

    printf("Dados guardados com sucesso!\n");
    return 1;
}

```

6.9 include/utils.h

```

/**
 * @file utils.h
 * @brief Funções auxiliares que uso em vários sítios.
 *
 * Leitura de inteiros e strings, validação de datas,
 * conversão para timestamp e limpeza do buffer do teclado.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

#ifndef UTILS_H
#define UTILS_H

/**
 * @brief Pede um número inteiro ao utilizador.
 * @param mensagem Texto que aparece no ecrã.
 * @return O número que o utilizador escreveu.
 */
int ler_inteiro(const char* mensagem);

/**
 * @brief Pede uma string ao utilizador (seguro, aceita espaços).
 * @param mensagem Texto que aparece no ecrã.
 * @param destino Onde guardar a string.
 * @param tamanho Tamanho máximo do array destino.
 */
void ler_string(const char* mensagem, char* destino, int tamanho);

/**
 * @brief Verifica se uma data está no formato DD/MM/AAAA.
 * @param data String com a data.
 * @return 1 se a data é válida, 0 se não é.
 */
int validar_data(const char* data);

/**
 * @brief Converte uma data DD/MM/AAAA para timestamp.
 * Timestamp = dias desde 01/01/2026.
 * @param data Data no formato DD/MM/AAAA.
 * @return Timestamp correspondente.
 */
int data_para_timestamp(const char* data);

/**
 * @brief Devolve a data actual (timestamp).
 * Nesta versão uso um valor fixo para simplificar.
 * @return Timestamp da data actual.
 */
int obter_data_atual(void);

/**

```

```

* @brief Limpa o que ficou pendente no buffer do teclado.
* É útil depois de usar scanf() para não haver leituras fantasmas.
*/
void limpar_buffer(void);

#endif /* UTILS_H */

```

6.10 src/utils.c

```

/**
 * @file utils.c
 * @brief Implementação das funções auxiliares.
 *
 * Código que realmente faz a leitura, validação de datas,
 * conversão e limpeza do buffer.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

#include <stdio.h>
#include <string.h>
#include "utils.h"

int ler_inteiro(const char* mensagem) {
    int valor;
    printf("%s", mensagem);
    scanf("%d", &valor);
    limpar_buffer(); /* tira o '\n' que ficou */
    return valor;
}

void ler_string(const char* mensagem, char* destino, int tamanho) {
    printf("%s", mensagem);
    fgets(destino, tamanho, stdin);
    destino[strcspn(destino, "\n")] = '\0'; /* tira a quebra de linha do fim */
}

int validar_data(const char* data) {
    int dia, mes, ano;

    /* a data deve ter mesmo 10 caracteres: DD/MM/AAAA */
    if (strlen(data) != 10) return 0;

    /* tenta extrair dia, mês e ano */
    if (sscanf(data, "%d/%d/%d", &dia, &mes, &ano) != 3) return 0;

    /* validações básicas */
    if (dia < 1 || dia > 31) return 0;
    if (mes < 1 || mes > 12) return 0;
    if (ano < 2026 || ano > 2100) return 0;

    return 1;
}

int data_para_timestamp(const char* data) {
    int dia, mes, ano;
    int dias_por_mes[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int timestamp = 0;

    sscanf(data, "%d/%d/%d", &dia, &mes, &ano);

    /* anos completos desde 2026 */

```

```

for (int y = 2026; y < ano; y++) {
    timestamp += 365;
}

/* meses completos do ano actual */
for (int m = 1; m < mes; m++) {
    timestamp += dias_por_mes[m - 1];
}

/* dias do mês actual (o dia 1 dá 0) */
timestamp += (dia - 1);

return timestamp;
}

int obter_data_atual(void) {
    /* Por simplicidade, uso um valor fixo.
       Numa versão mais completa podia usar time.h, mas
       para o E-Fólio isto é suficiente. */
    return 100;
}

/**
 * @brief Limpa o buffer do teclado
 *
 * Remove todos os caracteres pendentes até encontrar \n ou EOF.
 * util apos scanf para evitar leituras estranhas.
 */
void limpar_buffer(void) {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}

```

6.11 src/main.c

```

/**
 * @file main.c
 * @brief Programa principal do GreenTrack.
 *
 * Aqui é onde o programa começa. Mostro um menu, leio a opção
 * do utilizador e chamo as funções certas conforme o que ele
 * escolher. O ciclo repete até ele pedir para sair.
 *
 * @author Edgar Pinto Matuminy
 * @date abril 2026
 */

#include <stdio.h>
#include <stdlib.h>
#include "gestor_plantas.h"
#include "gestor_regas.h"
#include "gestor_tarefas.h"
#include "gestor_dados.h"
#include "utils.h"

/**
 * @brief Função principal do programa
 *
 * Inicializa todos os módulos, carrega os dados existentes,
 * exibe um menu interativo e processa as opções do utilizador.
 *
 * @return 0 se execução normal
 */
int main(void) {
    int opcao;
    int id, intervalo, data, quantidade;
}

```

```

char nome[MAX_NOME], especie[MAX_ESPECIE], data_plantio[MAX_DATA];
char descricao[MAX_DESCRICAO];

/* Inicializar tudo e carregar o que já estava guardado */
inicializar_plantas();
inicializar_regas();
inicializar_tarefas();
carregar_dados();

do {
    /* Mostrar o menu principal */

printf("=====
                                     \n");

    printf("\n===== Menu GREENTRACK
                                     \n");

printf("=====
                                     \n");

    printf("1. Listar plantas\n");
    printf("2. Adicionar planta\n");
    printf("3. Registrar rega\n");
    printf("4. Verificar necessidades de rega\n");
    printf("5. Listar tarefas pendentes\n");
    printf("6. Criar tarefa\n");
    printf("7. Concluir tarefa\n");
    printf("8. Guardar e sair\n");

printf("=====
                                     \n");

    printf("\n===== INTRODUIZ DADOS ABAIXO
                                     \n");

printf("=====
                                     \n");

    printf("Opcao: ");
    scanf("%d", &opcao);
    limpar_buffer(); /* Remove o \n pendente */

    /* Executar a opção escolhida */
    switch (opcao) {
        case 1:
            /* Listar todas as plantas */
            listar_plantas();
            break;

        case 2:
            printf("ID da planta: ");
            scanf("%d", &id);
            limpar_buffer();

            printf("Nome: ");
            ler_string("", nome, MAX_NOME);

            printf("Espécie: ");
            ler_string("", especie, MAX_ESPECIE);

            printf("Data de plantio (DD/MM/AAAA): ");
            ler_string("", data_plantio, MAX_DATA);

            printf("Intervalo de rega (dias): ");
            scanf("%d", &intervalo);

            adicionar_planta(id, nome, especie, data_plantio, intervalo, obter_data_atual());
            guardar_dados(); /* adiciona e guarda rapidamente para não perder nada */
            break;
    }
} while (opcao != 8);

```

```

case 3:
    printf("ID da planta: ");
    scanf("%d", &id);
    printf("Quantidade de água (ml): ");
    scanf("%d", &quantidade);
    registrar_rega(id, obter_data_atual(), quantidade);
    guardar_dados();
    break;

case 4:
    verificar_necessidade_rega(obter_data_atual());
    break;

case 5:
    listar_tarefas_pendentes();
    break;

case 6:
    /* Criar tarefa nova */
    printf("Descrição da tarefa: ");
    ler_string("", descricao, MAX_DESCRICA0);
    printf("Data prevista (timestamp): ");
    scanf("%d", &data);
    /* Cria a tarefa e guarda rapido */
    criar_tarefa(descricao, data);
    guardar_dados();
    break;

case 7:
    /* Concluir tarefa existente */
    printf("ID da tarefa: ");
    scanf("%d", &id);
    concluir_tarefa(id);
    guardar_dados();
    break;

case 8:
    guardar_dados();
    printf("Dados guardados. A sair...\n");
    break;

default:
    printf("Opção inválida!\n");
}
} while (opcao != 8); /* repete até o utilizador escolher sair */

return 0;
}

```

6.12 Makefile

```

# =====
# Makefile para o projeto GreenTrack
# Autor: Edgar Pinto Matuminy
# Data: abril 2026
#
# Comandos disponíveis:
# make      - compila o programa
# make clean - remove ficheiros temporários
# make run  - compila e executa
# =====

CC = gcc
CFLAGS = -Wall -Wextra -Wpedantic -I./include
SRCDIR = src

```

```
TARGET = greentrack.exe

OBJS = $(SRCDIR)/main.o \
      $(SRCDIR)/gestor_plantas.o \
      $(SRCDIR)/gestor_regas.o \
      $(SRCDIR)/gestor_tarefas.o \
      $(SRCDIR)/gestor_dados.o \
      $(SRCDIR)/utils.o

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)
    @echo "Compilação concluída! Executável: $(TARGET)"

$(SRCDIR)/%.o: $(SRCDIR)/%.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(SRCDIR)/*.o $(TARGET)
    @echo "Ficheiros temporários removidos."

run: $(TARGET)
    ./$(TARGET)

.PHONY: all clean run
```

7. Declaração de Autoria

Declaro que o presente trabalho foi desenvolvido por mim, Edgar Pinto Matuminy, com base nos excertos fornecidos no enunciado e nos materiais de estudo da unidade curricular (Recursos Formativos e Atividades Formativas). Todo o código foi escrito, testado e documentado por mim, sendo as fontes consultadas devidamente referenciadas ao longo do relatório.