

”

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



**UNIDADE CURRICULAR:** Laboratório de Programação

**CÓDIGO:** 21178

**DOCENTE:** Nelson Russo

**A preencher pelo estudante**

**NOME:** João Manuel Pacheco Seco Marques

**N.º DE ESTUDANTE:** 2403882

**CURSO:** Licenciatura de Engenharia Informática

**DATA DE ENTREGA:** 17 / 04 / 2026

## TRABALHO / RESOLUÇÃO:

1. Introdução .....	3
2. Tarefa 1: Análise e Correção de Erros.....	4
3. Tarefa 2: Reorganização Modular .....	8
3.1 Arquitectura e Estrutura de Módulos.....	8
3.2 Encapsulamento e Gestão de Dependências.....	9
3.3 Funções Públicas vs. Privadas .....	10
3.4 Estruturas de Dados .....	10
3.5 Alternativas Consideradas e Rejeitadas.....	10
3.6 Persistência Imediata e Geração de IDs .....	11
4. Conclusão .....	11
5. Anexo - Código Desenvolvido.....	12
5.1 include/tipos.h.....	12
5.2 include/plantas.h .....	13
5.3 src/plantas.c .....	14
5.4 include/regas.h .....	20
5.5 src/regas.c .....	20
5.6 include/tarefas.h.....	25
5.7 src/tarefas.c .....	26
5.8 include/io.h .....	31
5.9 src/io.c .....	31
5.10 src/main.c .....	34
5.11 Makefile.....	39

## 1. INTRODUÇÃO

Neste trabalho, o meu objetivo foi analisar, corrigir e reorganizar modularmente o sistema **GreenTrack**, uma plataforma de gestão de jardins comunitários que permite registrar plantas, controlar regas e gerir tarefas de manutenção, com persistência de dados em ficheiros CSV.

O código original que nos foi fornecido é um programa monolítico em C, onde todas as funcionalidades, estruturas de dados e variáveis globais coexistem num único ficheiro. Esta organização até funciona para fins didáticos iniciais, mas viola princípios fundamentais de engenharia de software: não há encapsulamento, o acoplamento entre componentes é excessivo e torna impossível reutilizar ou testar módulos de forma isolada.

A minha estratégia de resolução focou-se em três partes principais:

### 1. Identificação e correção sistemática de erros:

Analisei cada excerto individualmente, identificando erros de lógica, de gestão de ficheiros, de leitura de input e de violação de pré-condições. Para cada erro, documentei a categoria, as consequências em execução e a correção aplicada com a respetiva justificação.

### 2. Separação em módulos com responsabilidade clara:

Dividi o código monolítico em módulos coesos (plantas, regas, tarefas, io), cada um com responsabilidade bem definida. Os arrays globais originais passaram a ser declarados como static dentro dos respectivos módulos .c, sendo o acesso feito exclusivamente através de funções públicas expostas nos ficheiros .h. Desta forma, garanti um bom encapsulamento, impedindo que qualquer módulo aceda diretamente aos dados de outro.

### 3. Persistência imediata:

Como o enunciado exige, qualquer operação de escrita (adicionar planta, registrar rega, criar tarefa, concluir tarefa) desencadeia imediatamente a gravação dos dados em disco, garantindo consistência entre o estado em memória e os ficheiros CSV.

Todo o desenvolvimento seguiu o standard C99, compilado com GCC (MinGW).

## 2. TAREFA 1: ANÁLISE E CORREÇÃO DE ERROS

Durante a análise, identifiquei **18 erros** nos excertos fornecidos, que organizei em 7 grupos temáticos.

### Tipo de Grupos de Erro:

**A** - Violações de Limites de Arrays (Buffer Overflow)

**B** - Erros de Atribuição em Vez de Comparação (= em vez de ==)

**C** - Problemas de Gestão de Ficheiros e Persistência

**D** - Problemas de Input/Output

**E** - Lógica de Negócio: Falha na Atualização de ultima\_rega

**F** - Lógica de Negócio: Geração de IDs e Inicialização Incorreta

**G** - Validação de Dados Insuficiente e Design de Funções

A tabela abaixo resume cada erro com a respetiva localização, categoria, consequência em execução e correção aplicada.

Erro	Excerto	Categoria	Consequência	Correção e Justificação
<b>A1</b>	1	Violação de limites	Buffer overflow se plantas.csv > 100 registos; escrita em memória não reservada.	Verificar total_plantas < MAX_PLANTAS antes de cada inserção. Programação defensiva obrigatória em C. <a href="#">Ver plantas_carregar()</a> .
<b>A2</b>	3	Violação de limites	Se total_regas ≥ MAX_REGAS, escrita fora dos limites; comportamento indefinido.	Verificar total_regas < MAX_REGAS; retornar 0 se limite atingido. Ver <a href="#">regas_registar()</a> .
<b>A3</b>	5	Violação de limites	Buffer overflow se total_tarefas ≥ MAX_TAREFAS; corrupção de memória.	Verificar total_tarefas < MAX_TAREFAS; retornar int para indicar sucesso/falha. Ver <a href="#">tarefas_criar()</a> .

Erro	Excerto	Categoria	Consequência	Correção e Justificação
<b>A4</b>	2	Off-by-one	i <= total_plantas accede posição não inicializada; imprime lixo de memória.	Substituir por i < total_plantas. Índices válidos em C: [0, n-1]. <a href="#">Ver plantas_listar()</a> .
<b>B1</b>	6	Atribuição vs comparação	concluida = 0 é atribuição: (1) bloco if nunca executa; (2) reverte tarefas concluídas para pendentes.	Substituir = por ==: if (concluida == 0). Ver <a href="#">tarefas_listar_pendentes()</a> .
<b>B2</b>	7	Atribuição vs comparação	id_tarefa = id sobrescreve o ID e avalia como verdadeiro se id ≠ 0; conclui sempre a 1.ª tarefa, corrompendo o ID original. Erro grave: corrompe dois campos.	Substituir = por ==: if (id_tarefa == id). Ver <a href="#">tarefas_concluir()</a> .
<b>C1</b>	1	Gestão de ficheiros / Lógica	feof() só devolve verdadeiro após falha de leitura; ciclo executa uma iteração a mais, duplicando o último registo.	Avaliar retorno de fscanf: while (fscanf(...) == N). Pára imediatamente ao falhar. Ver <a href="#">plantas_carregar()</a> , <a href="#">regas_carregar()</a> , <a href="#">tarefas_carregar()</a> .
<b>C2</b>	1	Gestão de ficheiros / Validação	f_regas aberto sem verificação NULL; se ficheiro não existe, feof(NULL) causa segfault.	Adicionar if (f == NULL) após cada fopen. Ver <a href="#">regas_carregar()</a> , <a href="#">tarefas_carregar()</a> .

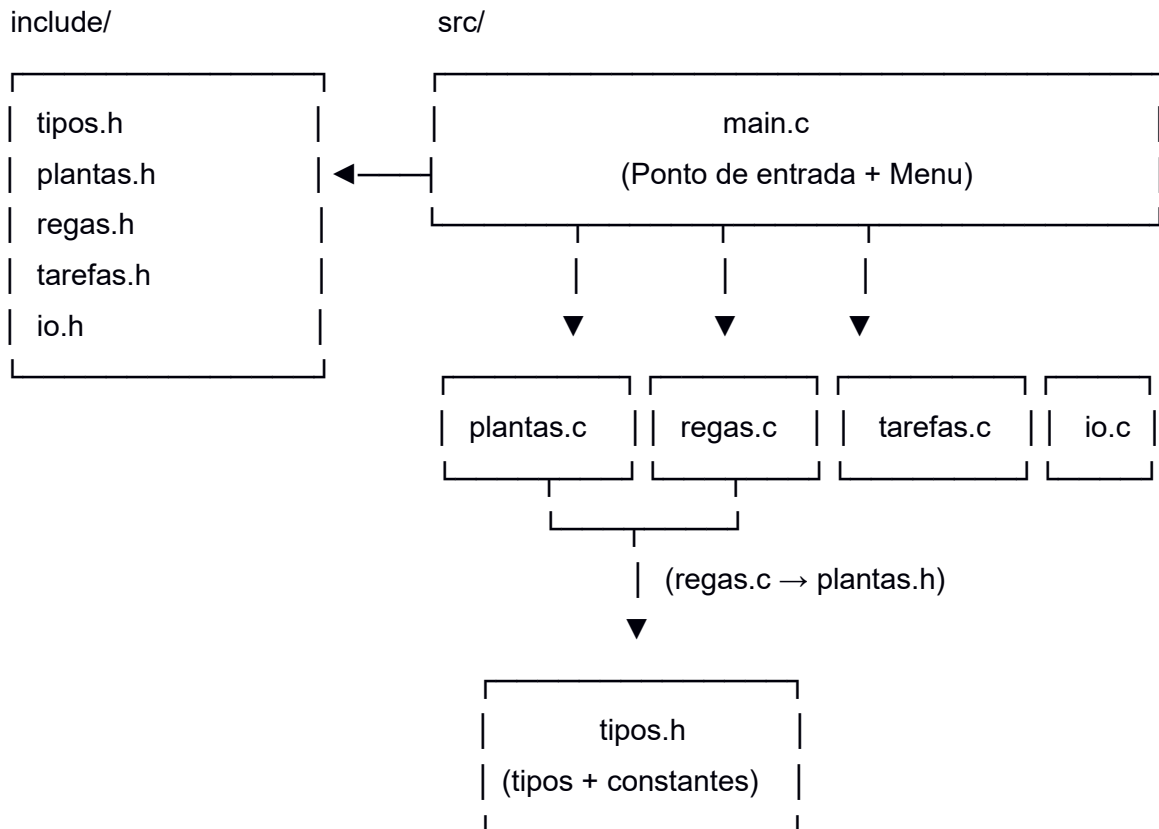
Erro	Excerto	Categoria	Consequência	Correção e Justificação
<b>C3</b>	8	Persistência incompleta	Só guarda plantas; regas e tarefas nunca persistidos. Perda total de dados ao reiniciar.	Adicionar gravação de regas.csv e tarefas.csv. Cada módulo passou a ter função de gravação própria.
<b>C4</b>	9	Falta persistência imediata	Após adicionar planta, não invoca gravação; dados perdidos se programa termina inesperadamente. Mesmo em registrar_regas() e criar_tarefa().	Invocar gravação imediata após cada escrita no array. Ver <a href="#">plantas adicionar()</a> , <a href="#">regas registrar()</a> , <a href="#">tarefas criar()</a> .
<b>D1</b>	10	Leitura de strings	scanf("%s") lê só até ao 1.º espaço; "Rosa Chinesa" fica "Rosa". Resto corrompe leituras seguintes.	Usar fgets para ler strings com espaços. Ver <a href="#">io ler string() em io.c</a> .
<b>D2</b>	10	Buffer de input residual	Após scanf("%d"), \n fica no buffer; próxima leitura textual (fgets) consome \n e retorna vazio.	Limpar buffer com while(getchar() != '\n'). Nunca usar fflush(stdin). Ver <a href="#">io limpar buffer()</a> .
<b>E1</b>	3	Lógica de negócio	Não atualiza ultima_regas da planta; verificar_regas() indica falsamente que precisa de regas. Regra explícita do enunciado.	Após registrar regas, chamar plantas_atualizar_ultima_regas(i d_planta, data). Ver <a href="#">regas registrar()</a> .

Erro	Excerto	Categoria	Consequência	Correção e Justificação
<b>F1</b>	9	Lógica de negócio	Recebe id como parâmetro, violando regra do enunciado: IDs devem ser gerados automaticamente. Permite IDs duplicados.	Remover parâmetro id; gerar ID internamente via <code>proximo_id_planta</code> . Ver <a href="#">plantas_adicionar()</a> .
<b>F2</b>	3, 5, 9	Lógica de negócio / Dados	<code>id = total_X + 1</code> assume IDs sequenciais; se CSV tem IDs não sequenciais, podem gerar-se IDs duplicados.	Após carregamento, calcular <code>proximo_id = max_id + 1</code> . Ver <code>calcular_proximo_id()</code> em cada módulo. <a href="#">plantas.c</a> , <a href="#">regas.c</a> , <a href="#">tarefas.c</a>
<b>F3</b>	9	Lógica de negócio	<code>ultima_rega = 0</code> (01/01/2026), mas comentário diz "data atual". Planta sinalizada como precisando de rega imediata.	Inicializar <code>ultima_rega</code> com <code>data_atual</code> passada como parâmetro. Ver <a href="#">plantas_adicionar()</a> .
<b>G1</b>	3	Validação de dados	Aceita qualquer <code>id_planta</code> sem verificar existência; cria regas órfãs para plantas inexistentes.	Validar <code>id_planta</code> via <code>plantas_obter_por_id()</code> antes de registrar. Ver <a href="#">regas_registar()</a> .
<b>G2</b>	4	Design de função	Função void só imprime; impossível reutilizar para contagem ou automação. Mistura lógica com apresentação.	Retornar int (n.º de plantas que precisam de rega). Ver <a href="#">plantas_verificar_rega()</a> .

### 3. TAREFA 2: REORGANIZAÇÃO MODULAR

#### 3.1 Arquitectura e Estrutura de Módulos

A arquitetura e o encapsulamento físico são ilustrados no seguinte diagrama:



Optei por dividir o programa em 5 módulos e 1 ponto de entrada (10 ficheiros `.h/.c`), separando as interfaces públicas ( `include/` ) das implementações privadas ( `src/` ). Para gerir a compilação modular, criei um Makefile utilizando a flag `-I./include`

Módulo	Ficheiros	Responsabilidade
<b>tipos</b>	<code>include/tipos.h</code>	Estruturas (Planta, Rega, Tarefa) e constantes ( <code>MAX_*</code> )
<b>plantas</b>	<code>include/plantas.h</code> <code>src/plantas.c</code>	CRUD e persistência do array de plantas
<b>regas</b>	<code>include/regas.h</code> <code>src/regas.c</code>	CRUD e persistência do array de regas



Módulo	Ficheiros	Responsabilidade
<b>tarefas</b>	include/tarefas.h src/tarefas.c	CRUD e persistência do array de tarefas
<b>io</b>	include/io.h src/io.c	Leitura segura de input e formatação de saída
<b>main</b>	src/main.c	Ponto de entrada, menu interativo, orquestração

### 3.2 Encapsulamento e Gestão de Dependências

O requisito mais exigente do enunciado era garantir encapsulamento. Declarei assim cada array de dados (plantas, regas, tarefas) e os seus contadores (total\_X, proximo\_id\_X) como static nos respetivos ficheiros .c. Assim, a sua visibilidade fica restrita à implementação, bloqueando qualquer acesso externo direto.

A árvore de inclusões lógicas que organiza a comunicação é a seguinte:

main.c → plantas.h → tipos.h	regas.c → plantas.h (validação + ultima_rega)
→ regas.h → tipos.h	→ tipos.h
→ tarefas.h → tipos.h	plantas.c → tipos.h (sem dependências cruzadas)
→ io.h	tarefas.c → tipos.h (sem dependências cruzadas)
	io.c → (independente)

A única dependência cruzada no sistema é regas.c → plantas.h.

Resolvi-a exclusivamente através da chamada a funções públicas ( plantas\_obter\_por\_id() e plantas\_atualizar\_ultima\_rega() ), sem nunca aceder diretamente ao array global de plantas. A dependência é estritamente unidirecional, o que me pareceu a abordagem mais limpa.

### 3.3 Funções Públicas vs. Privadas

- **Módulo plantas:**

**10 Públicas** (inicializar, carregar, guardar, adicionar, listar, obter\_por\_indice, obter\_por\_id, atualizar\_ultima\_rega, total, verificar\_rega).

**1 Privada** static (calcular\_proximo\_id).

- **Módulo regas:**

**7 Públicas** (inicializar, carregar, guardar, registrar, listar, obter\_por\_indice, total).

**1 Privada** static (calcular\_proximo\_id).

- **Módulo tarefas:**

**9 Públicas** (inicializar, carregar, guardar, criar, concluir, listar\_pendentes, listar\_todas, obter\_por\_indice, total).

**1 Privada** static (calcular\_proximo\_id).

- **Módulo io:**

**6 Públicas** (ler\_string, ler\_inteiro, limpar\_buffer, mostrar\_menu, pausa, eof).

**Nenhuma função privada.**

### 3.4 Estruturas de Dados

Cada módulo gere um array estático de tamanho fixo, conforme os limites definidos pelo enunciado (MAX\_PLANTAS, MAX\_REGAS, MAX\_TAREFAS). Optei por manter arrays estáticos porque respeita o âmbito didático exigido, mantém a simplicidade na gestão da memória e garante segurança no acesso linear aos dados.

### 3.5 Alternativas Consideradas e Rejeitadas

- **Alternativa 1: Módulo de dados central** (dados.h/dados.c) - Agrupar os três arrays num único ficheiro de estado global.

**Rejeição:** Viola o princípio de alta coesão e introduz acoplamento global, contradizendo o princípio de mínimo privilégio.

- **Alternativa 2: Alocação dinâmica de memória** - Utilizar malloc/realloc para criar arrays expansíveis.

**Rejeição:** O enunciado impõe os limites MAX\_\* como constrangimento explícito. Adicionar gestão dinâmica traria complexidade desnecessária (memory leaks, gestão de ponteiros) sem valorizar os requisitos do domínio (Módulos 1 e 2).

- **Decisão adotada:** Separação por domínio (Plantas, Regas, Tarefas) com instâncias static, permitindo um equilíbrio ideal entre encapsulamento, interface contratual limpa e persistência modular.

### 3.6 Persistência Imediata e Geração de IDs

Para satisfazer o requisito de que os dados persistidos "refletem imediatamente a alteração", todas as operações de escrita (adicionar, registrar, concluir) disparam imediatamente a função local guardar() do respetivo módulo (estratégia write-through). Paralelamente, em vez de assumir que o próximo ID é total + 1 (Erro F2), cada módulo calcula de forma autónoma  $\text{proximo\_id} = \text{max\_id} + 1$  no ato do carregamento, mitigando o risco de colisões em caso de CSVs não sequenciais.

## 4. CONCLUSÃO

O código original do GreenTrack mostrava problemas típicos: variáveis globais acessíveis por qualquer função, ausência de validação de limites, confusão entre atribuição e comparação, e gestão deficiente de ficheiros e input. Cada um destes problemas tem consequências reais em execução, desde dados corrompidos e comportamento indefinido até crashes por acesso ilegal à memória.

Foi um desafio lidar com código legado que dependia de variáveis globais. A introdução do encapsulamento via static nos módulos .c e o acesso exclusivo via funções públicas nos .h resolveu o problema do acesso global de forma limpa e idiomática em C. Cada módulo tornou-se dono exclusivo dos seus dados, e as interações passaram a ser mediadas por interfaces bem definidas, como no caso de regas.c → plantas.h, onde a atualização de ultima\_rega é feita via plantas\_atualizar\_ultima\_rega(), sem acesso direto ao array. A persistência imediata elimina a classe de bugs em que o utilizador assume que os dados foram gravados, mas o programa terminou antes da escrita.

A separação em módulos ajudou-me bastante a perceber o fluxo do programa. Quando cada módulo tem uma responsabilidade clara, fica muito mais fácil raciocinar sobre o que cada parte faz e onde procurar quando algo corre mal.

Se fosse a continuar a desenvolver este projeto, consideraria: alocação dinâmica (para remover limites fixos), um formato de ficheiro mais robusto (com validação de integridade) e testes unitários por função pública.

No entanto, dentro dos requisitos do enunciado e do standard C99, sinto que a solução apresentada equilibra bem rigor técnico, clareza de código e funcionalidade completa.

## 5. ANEXO - CÓDIGO DESENVOLVIDO

### 5.1 include/tipos.h

```
/* tipos.h - tipos e constantes do GreenTrack */

#ifndef TIPOS_H
#define TIPOS_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* — Tipos de dados ————— */

#define MAX_PLANTAS 100
#define MAX_REGAS 500
#define MAX_TAREFAS 200

typedef struct {
    // int id_antigo; // tirei isto, era antes de corrigir o F1
    int id;
    char nome[50];
    char especie[50];
    char data_plantio[11]; /* DD/MM/AAAA */
    int intervalo_rega; /* em dias */
    int ultima_rega; /* timestamp: dias desde 01/01/2026 */
} Planta;

typedef struct {
    int id_rega;
    int id_planta;
    int data_rega; /* timestamp: dias desde 01/01/2026 */
    int quantidade_agua; /* em ml */
} Rega;

typedef struct {
    int id_tarefa;
    char descricao[100];
    int data_prevista; /* timestamp: dias desde 01/01/2026 */
    int concluida; /* 0 = pendente | 1 = concluida */
} Tarefa;

#endif /* TIPOS_H */
```

## 5.2 include/plantas.h

```
/* plantas.h - prototipos do modulo de plantas */

#ifndef PLANTAS_H
#define PLANTAS_H

#include "tipos.h"

// funcoes a exportar

void plantas_inicializar(void);
int plantas_carregar(const char *nome_ficheiro);
int plantas_guardar(const char *nome_ficheiro);

// Retirei o ID dos parametros (erro F1) e meti a data_atual (erro F3)
int plantas_adicionar(const char *nome, const char *especie,
                     const char *data_plantio, int intervalo,
                     int data_atual);

// void plantas_adicionar(int id, ...); // versao original antes de corrigir
// F1/F3

void plantas_listar(void);
int plantas_obter_por_indice(int indice, Planta *destino);
int plantas_obter_por_id(int id, Planta *destino);
int plantas_atualizar_ultima_rega(int id_planta, int data_rega);
int plantas_total(void);

// Passei isto a retornar int em vez de void para contar as plantas (erro G2)
// void plantas_verificar_rega(int data_atual); // versao antiga
int plantas_verificar_rega(int data_atual);

#endif /* PLANTAS_H */
```

### 5.3 src/plantas.c

```
/* =====
 * plantas.c – Implementação do módulo de Plantas
 * GreenTrack – Gestão de Jardins Comunitários
 *
 * Encapsulamento: os arrays e contadores são static, invisíveis
 * fora deste ficheiro. O acesso faz-se exclusivamente pelas
 * funções públicas declaradas em plantas.h.
 *
 * Notas sobre correções que apliquei:
 * - A1: Tive de verificar os limites do array no carregamento,
 *   senão o programa podia rebentar se houvesse mais de 100 plantas.
 * - A4: No listar, corriji o i <= total_plantas para i < total_plantas,
 *   porque em C os índices válidos vão de 0 a n-1.
 * - C1: Troquei o feof() por fscanf() no ciclo – o feof só devolve
 *   true depois de uma leitura falhar, o que duplicava o último registo.
 * - C4: Adicionei persistência imediata – guardar logo após alterar.
 * - F1: O ID agora é gerado automaticamente, não vem do utilizador.
 * - F2: Os IDs agora são max_id+1 em vez de total+1, para evitar
 *   duplicações se o CSV tiver IDs não sequenciais.
 * - F3: ultima_rega passa a ser a data atual e não zero.
 * - G2: verificar_rega() agora retorna o número de plantas que
 *   precisam de rega em vez de ser void.
 * ===== */

#include "plantas.h"

/* — Dados privados (encapsulados) ————— */

static Planta plantas[MAX_PLANTAS];
static int total_plantas = 0;
static int proximo_id_planta = 1;

/* — Funções privadas (static) ————— */

/* Calcula o maior ID existente no array e define proximo_id_planta */
static void calcular_proximo_id(void)
{
    int max_id = 0;
    int i;

    for (i = 0; i < total_plantas; i++)
    {
        if (plantas[i].id > max_id)
        {
            max_id = plantas[i].id;
        }
    }
}
```

```

    proximo_id_planta = max_id + 1;
}

/* — Funções públicas — */

void plantas_inicializar(void)
{
    total_plantas = 0;
    proximo_id_planta = 1;
}

/* Remodelei esta função para resolver o erro C1 (feof duplicava o
último registo) e o erro A1 (sem verificação de limites podia
haver buffer overflow se o CSV tivesse mais de 100 registos). */
int plantas_carregar(const char *nome_ficheiro)
{
    FILE *f;
    int campos_lidos;

    f = fopen(nome_ficheiro, "r");
    if (f == NULL)
    {
        /* Ficheiro não existe — não é erro, array fica vazio */
        return 0;
    }

    while (total_plantas < MAX_PLANTAS)
    {
        campos_lidos = fscanf(f, "%d,%49[^\n],%49[^\n],%10[^\n],%d,%d\n",
                                &plantas[total_plantas].id,
                                plantas[total_plantas].nome,
                                plantas[total_plantas].especie,
                                plantas[total_plantas].data_plantio,
                                &plantas[total_plantas].intervalo_rega,
                                &plantas[total_plantas].ultima_rega);

        if (campos_lidos != 6)
        {
            break;
        }
        total_plantas++;
    }

    if (!feof(f))
    {
        printf("AVISO: Limite de %d plantas atingido. Registos adicionais "
               "foram ignorados.\n", MAX_PLANTAS);
    }
}

```

```

fclose(f);

/* Correção F2: calculei o próximo ID com base no maior ID existente
   em vez de usar total+1, porque se o CSV tiver buracos nos IDs
   podia criar IDs duplicados */
calcular_proximo_id();

return 1;
}

int plantas_guardar(const char *nome_ficheiro)
{
    FILE *f;
    int i;

    f = fopen(nome_ficheiro, "w");
    if (f == NULL)
    {
        printf("Erro ao guardar plantas em %s\n", nome_ficheiro);
        return 0;
    }

    for (i = 0; i < total_plantas; i++)
    {
        fprintf(f, "%d,%s,%s,%s,%d,%d\n",
                plantas[i].id,
                plantas[i].nome,
                plantas[i].especie,
                plantas[i].data_plantio,
                plantas[i].intervalo_rega,
                plantas[i].ultima_rega);
    }

    fclose(f);
    return 1;
}

/* Corrigi esta função para resolver os erros A2 (verificar limites),
   F1 (ID automático), F3 (ultima_rega = data_atual) e C4 (persistência
   imediata). No início tinha uma versão que não testava o limite e
   o ID vinha do parâmetro – depois de ver os erros percebi que tinha
   de reestruturar tudo. */
int plantas_adicionar(const char *nome, const char *especie,
                     const char *data_plantio, int intervalo,
                     int data_atual)
{
    if (total_plantas >= MAX_PLANTAS)

```



```

{
    printf("Erro: limite de plantas atingido (%d).\n", MAX_PLANTAS);
    return 0;
}

/* if (total_plantas == MAX_PLANTAS) return; – Versão antiga que não
   dava mensagem de erro nem retornava 0, o chamador não sabia que
   falhou. A condição >= funciona melhor (erro A2). */

plantas[total_plantas].id = proximo_id_planta;
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;

/* printf("DEBUG: adicionada planta com id %d\n", proximo_id_planta); */

total_plantas++;
proximo_id_planta++;

plantas_guardar("plantas.csv");

return 1;
}

/* Corrigi o erro A4 aqui – o ciclo original tinha i <= total_plantas
   e acedia a uma posição não inicializada (lixo de memória). */
void plantas_listar(void)
{
    int i;

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

    if (total_plantas == 0)
    {
        printf("Nenhuma planta registrada.\n");
        return;
    }

    for (i = 0; i < total_plantas; i++)
    {
        printf("ID: %d | Nome: %s | Espécie: %s | Plantio: %s | "
               "Intervalo: %d dias | Última rega: dia %d\n",
               plantas[i].id,
               plantas[i].nome,
               plantas[i].especie,
               plantas[i].data_plantio,

```

```

        plantas[i].intervalo_rega,
        plantas[i].ultima_rega);
    }
}

int plantas_obter_por_indice(int indice, Planta *destino)
{
    if (indice < 0 || indice >= total_plantas)
    {
        return 0;
    }

    *destino = plantas[indice];
    return 1;
}

int plantas_obter_por_id(int id, Planta *destino)
{
    int i;

    for (i = 0; i < total_plantas; i++)
    {
        if (plantas[i].id == id)
        {
            *destino = plantas[i];
            return i;
        }
    }

    return -1;
}

int plantas_atualizar_ultima_rega(int id_planta, int data_rega)
{
    int i;

    for (i = 0; i < total_plantas; i++)
    {
        if (plantas[i].id == id_planta)
        {
            plantas[i].ultima_rega = data_rega;

            /* Guardar logo para não perder dados se o programa
               fechar inesperadamente (correção C4) */
            plantas_guardar("plantas.csv");
            return 1;
        }
    }
}

```

```

    return 0;
}

int plantas_total(void)
{
    return total_plantas;
}

/* Mudei o tipo de retorno de void para int para resolver o erro G2 –
   a função original só imprimia e não retornava nada, o que torna
   impossível reutilizar para contagem ou automação. */
int plantas_verificar_rega(int data_atual)
{
    int i;
    int dias;
    int precisam = 0;

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

    for (i = 0; i < total_plantas; i++)
    {
        dias = data_atual - plantas[i].ultima_rega;

        if (dias >= plantas[i].intervalo_rega)
        {
            printf("Planta %s (ID: %d) precisa de rega! "
                   "(ultima: %d dias atras)\n",
                   plantas[i].nome, plantas[i].id, dias);
            precisam++;
        }
    }

    if (precisam == 0)
    {
        printf("Todas as plantas estao regadas.\n");
    }

    return precisam;
}

```

## 5.4 include/regas.h

```
/* regas.h - prototipos do modulo de regas */

#ifndef REGAS_H
#define REGAS_H

#include "tipos.h"

// prototipos

void regas_inicializar(void);
int regas_carregar(const char *nome_ficheiro);
int regas_guardar(const char *nome_ficheiro);

// Tive de por validacao da planta (erro G1) e atualizar ultima_rega (erro E1)
// int regas_registar(int id_planta, int data, int quantidade); // assinatura
// igual, mas agora valida
int regas_registar(int id_planta, int data, int quantidade);

void regas_listar(void);
int regas_obter_por_indice(int indice, Rega *destino);
int regas_total(void);

#endif /* REGAS_H */
```

## 5.5 src/regas.c

```
/* =====
 * regas.c – Implementação do módulo de Regas
 * GreenTrack – Gestão de Jardins Comunitários
 *
 * Encapsulamento: os arrays e contadores são static, invisíveis
 * fora deste ficheiro. O acesso faz-se exclusivamente pelas
 * funções públicas declaradas em regas.h.
 *
 * Este módulo depende de plantas.h para validar a existência da
 * planta e atualizar o campo ultima_rega.
 *
 * Notas sobre correções que apliquei:
 * - A2: Verificação de limites no array de regas antes de inserir.
 * - C1: Troquei feof() por fscanf() no ciclo de leitura.
 * - C2: Adicionei verificação de NULL após fopen – se o ficheiro
 * não existe, o programa não rebenta com segfault.
 * - C4: Persistência imediata após cada rega registada.
 * - E1: Após registar a rega, atualizo o ultima_rega da planta
 * através de plantas_atualizar_ultima_rega().
 * - F2: ID gerado como max_id+1 em vez de total+1.
 */
```

```

* - G1: Valido se a planta existe antes de criar a rega, senão
*   ficavam registos órfãos de plantas que não existem.
* ===== */

#include "regas.h"
#include "plantas.h"

/* — Dados privados (encapsulados) — */

static Rega regas[MAX_REGAS];
static int total_regas = 0;
static int proximo_id_rega = 1;

/* — Funções privadas (static) — */

static void calcular_proximo_id(void)
{
    int max_id = 0;
    int i;

    for (i = 0; i < total_regas; i++)
    {
        if (regas[i].id_rega > max_id)
        {
            max_id = regas[i].id_rega;
        }
    }
    proximo_id_rega = max_id + 1;
}

/* — Funções públicas — */

void regas_inicializar(void)
{
    total_regas = 0;
    proximo_id_rega = 1;
}

/* O erro C2 estava aqui – o fopen era feito sem verificar se
   resultava em NULL, e se o ficheiro não existia o programa
   ia tentar ler de um ponteiro inválido (segfault). */
int regas_carregar(const char *nome_ficheiro)
{
    FILE *f;
    int campos_lidos;

    f = fopen(nome_ficheiro, "r");
    if (f == NULL)

```

```

{
    return 0;
}

/* Também corriji aqui o C1 – usei fscanf em vez de feof
   porque o feof só sinaliza EOF depois de uma leitura falhar */
while (total_regas < MAX_REGAS)
{
    campos_lidos = fscanf(f, "%d,%d,%d,%d\n",
                          &regas[total_regas].id_rega,
                          &regas[total_regas].id_planta,
                          &regas[total_regas].data_rega,
                          &regas[total_regas].quantidade_agua);

    if (campos_lidos != 4)
    {
        break;
    }
    total_regas++;
}

if (!feof(f))
{
    printf("AVISO: Limite de %d regas atingido. Registos adicionais "
           "foram ignorados.\n", MAX_REGAS);
}

fclose(f);
calcular_proximo_id();

return 1;
}

int regas_guardar(const char *nome_ficheiro)
{
    FILE *f;
    int i;

    f = fopen(nome_ficheiro, "w");
    if (f == NULL)
    {
        printf("Erro ao guardar regas em %s\n", nome_ficheiro);
        return 0;
    }

    for (i = 0; i < total_regas; i++)
    {
        fprintf(f, "%d,%d,%d,%d\n",

```

```

        regas[i].id_rega,
        regas[i].id_planta,
        regas[i].data_rega,
        regas[i].quantidade_agua);
    }

    fclose(f);
    return 1;
}

/* Reestruturei esta função para resolver os erros A2 (limites),
G1 (validar planta), E1 (atualizar ultima_rega), F2 (ID automático)
e C4 (persistência imediata). Aproveitei e corriji o G1, validando
se a planta existe antes de avançar, senão criava registros órfãos. */
int regas_registar(int id_planta, int data, int quantidade)
{
    Planta planta_tmp;

    if (total_regas >= MAX_REGAS)
    {
        printf("Erro: limite de regas atingido (%d).\n", MAX_REGAS);
        return 0;
    }

    /* if (total_regas >= MAX_REGAS) return 0; – Tentei primeiro só
    retornar 0 sem mensagem, mas depois percebi que o utilizador
    precisa de saber porque é que falhou */

    if (plantas_obter_por_id(id_planta, &planta_tmp) == -1)
    {
        printf("Erro: planta com ID %d nao existe.\n", id_planta);
        return 0;
    }

    regas[total_regas].id_rega = proximo_id_rega;
    regas[total_regas].id_planta = id_planta;
    regas[total_regas].data_rega = data;
    regas[total_regas].quantidade_agua = quantidade;

    /* printf("DEBUG: rega id=%d para planta id=%d\n", proximo_id_rega,
    id_planta); */

    total_regas++;
    proximo_id_rega++;

    /* Atualizar ultima_rega da planta – erro E1 dizia que isto
    faltava no código original e fazia com que a verificação
    de regas indicasse falsamente que a planta precisava de rega */

```

```

    plantas_atualizar_ultima_rega(id_planta, data);

    regas_guardar("regas.csv");

    return 1;
}

void regas_listar(void)
{
    int i;
    Planta planta_tmp;

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

    if (total_regas == 0)
    {
        printf("Nenhuma rega registrada.\n");
        return;
    }

    for (i = 0; i < total_regas; i++)
    {
        printf("ID Rega: %d | Planta ID: %d | Data: dia %d | Agua: %d ml",
            regas[i].id_rega,
            regas[i].id_planta,
            regas[i].data_rega,
            regas[i].quantidade_agua);

        /* Mostrar nome da planta se existir */
        if (plantas_obter_por_id(regas[i].id_planta, &planta_tmp) != -1)
        {
            printf(" (%s)", planta_tmp.nome);
        }
        printf("\n");
    }
}

int regas_obter_por_indice(int indice, Rega *destino)
{
    if (indice < 0 || indice >= total_regas)
    {
        return 0;
    }

    *destino = regas[indice];
    return 1;
}

```



```
int regas_total(void)
{
    return total_regas;
}
```

## 5.6 include/tarefas.h

```
/* tarefas.h - prototipos do modulo de tarefas */

#ifndef TAREFAS_H
#define TAREFAS_H

#include "tipos.h"

// prototipos

void tarefas_inicializar(void);
int tarefas_carregar(const char *nome_ficheiro);
int tarefas_guardar(const char *nome_ficheiro);

// ID gerado automaticamente (erro F2), nao vem como parametro
int tarefas_criar(const char *descricao, int data_prevista);

// 0 == em vez de = ja foi corrigido (erro B2)
int tarefas_concluir(int id_tarefa);

void tarefas_listar_pendentes(void);
void tarefas_listar_todas(void);
int tarefas_obter_por_indice(int indice, Tarefa *destino);
int tarefas_total(void);

#endif /* TAREFAS_H */
```

## 5.7 src/tarefas.c

```
/* =====
 * tarefas.c – Implementação do módulo de Tarefas
 * GreenTrack – Gestão de Jardins Comunitários
 *
 * Encapsulamento: os arrays e contadores são static, invisíveis
 * fora deste ficheiro. O acesso faz-se exclusivamente pelas
 * funções públicas declaradas em tarefas.h.
 *
 * Notas sobre correções que apliquei:
 * - A3: Verificação de limites no array antes de criar tarefa.
 * - B1: No listar_pendentes, troquei = por == na comparação
 * com concluída – a versão original atribuía 0 em vez de
 * comparar, o que revertia tarefas concluídas para pendentes.
 * - B2: No concluir, troquei = por == – a versão original
 * sobrescrevia o ID da tarefa e ainda a marcava sempre como
 * concluída. Dois erros num só if.
 * - C1: Troquei feof() por fscanf() no ciclo de leitura.
 * - C4: Persistência imediata após criar e concluir tarefa.
 * - F2: ID gerado como max_id+1 em vez de total+1.
 * ===== */

#include "tarefas.h"

/* — Dados privados (encapsulados) ————— */

static Tarefa tarefas[MAX_TAREFAS];
static int total_tarefas = 0;
static int proximo_id_tarefa = 1;

/* — Funções privadas (static) ————— */

static void calcular_proximo_id(void)
{
    int max_id = 0;
    int i;

    for (i = 0; i < total_tarefas; i++)
    {
        if (tarefas[i].id_tarefa > max_id)
        {
            max_id = tarefas[i].id_tarefa;
        }
    }
    proximo_id_tarefa = max_id + 1;
}

/* — Funções públicas ————— */
```

```

void tarefas_inicializar(void)
{
    total_tarefas = 0;
    proximo_id_tarefa = 1;
}

/* Corrigi o erro C1 aqui – usei fscanf em vez de feof, e também
verifiquei se o ficheiro abriu corretamente (erro C2). */
int tarefas_carregar(const char *nome_ficheiro)
{
    FILE *f;
    int campos_lidos;

    f = fopen(nome_ficheiro, "r");
    if (f == NULL)
    {
        return 0;
    }

    while (total_tarefas < MAX_TAREFAS)
    {
        campos_lidos = fscanf(f, "%d,%99[^\n],%d,%d\n",
                                &tarefas[total_tarefas].id_tarefa,
                                &tarefas[total_tarefas].descricao,
                                &tarefas[total_tarefas].data_prevista,
                                &tarefas[total_tarefas].concluida);

        if (campos_lidos != 4)
        {
            break;
        }
        total_tarefas++;
    }

    if (!feof(f))
    {
        printf("AVISO: Limite de %d tarefas atingido. Registos adicionais "
               "foram ignorados.\n", MAX_TAREFAS);
    }

    fclose(f);
    calcular_proximo_id();

    return 1;
}

int tarefas_guardar(const char *nome_ficheiro)

```

```

{
    FILE *f;
    int i;

    f = fopen(nome_ficheiro, "w");
    if (f == NULL)
    {
        printf("Erro ao guardar tarefas em %s\n", nome_ficheiro);
        return 0;
    }

    for (i = 0; i < total_tarefas; i++)
    {
        fprintf(f, "%d,%s,%d,%d\n",
            tarefas[i].id_tarefa,
            tarefas[i].descricao,
            tarefas[i].data_prevista,
            tarefas[i].concluida);
    }

    fclose(f);
    return 1;
}

/* Corrige o erro A3 (verificar limites) e F2 (ID automático).
   Adicionei persistência imediata (C4) – guardei logo após criar
   a tarefa em vez de deixar para o fim do programa. */
int tarefas_criar(const char *descricao, int data_prevista)
{
    if (total_tarefas >= MAX_TAREFAS)
    {
        printf("Erro: limite de tarefas atingido (%d).\n", MAX_TAREFAS);
        return 0;
    }

    tarefas[total_tarefas].id_tarefa = proximo_id_tarefa;
    strcpy(tarefas[total_tarefas].descricao, descricao);
    tarefas[total_tarefas].data_prevista = data_prevista;
    tarefas[total_tarefas].concluida = 0;

    total_tarefas++;
    proximo_id_tarefa++;

    tarefas_guardar("tarefas.csv");

    return 1;
}

```

```

/* O erro B2 era aqui – o código original tinha if (id_tarefa = id)
em vez de ==. Isto fazia duas coisas erradas: sobrescrevia o ID
da tarefa e avaliava sempre como verdadeiro (se id ≠ 0), pelo
que concluía sempre a primeira tarefa que encontrava. */
int tarefas_concluir(int id_tarefa)
{
    int i;

    for (i = 0; i < total_tarefas; i++)
    {
        if (tarefas[i].id_tarefa == id_tarefa)
        {
            tarefas[i].concluida = 1;

            /* Guardar logo – se o programa fechar a meio, não
            perdemos a alteração */
            tarefas_guardar("tarefas.csv");
            return 1;
        }
    }

    printf("Erro: tarefa com ID %d nao encontrada.\n", id_tarefa);
    return 0;
}

/* O erro B1 estava aqui – o código original tinha if (concluida = 0)
em vez de ==. Isto era uma atribuição e não uma comparação:
punha concluida a 0 e o if era sempre verdadeiro, pelo que
listava todas as tarefas (mesmo as concluídas) como pendentes. */
void tarefas_listar_pendentes(void)
{
    int i;
    int encontrou = 0;

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

    for (i = 0; i < total_tarefas; i++)
    {
        if (tarefas[i].concluida == 0)
        {
            printf("Tarefa %d: %s (prevista: dia %d)\n",
                tarefas[i].id_tarefa,
                tarefas[i].descricao,
                tarefas[i].data_prevista);
            encontrou = 1;
        }
    }
}

```

```

    if (!encontrou)
    {
        printf("Nenhuma tarefa pendente.\n");
    }
}

void tarefas_listar_todas(void)
{
    int i;

    printf("=== TODAS AS TAREFAS ===\n");

    if (total_tarefas == 0)
    {
        printf("Nenhuma tarefa registrada.\n");
        return;
    }

    for (i = 0; i < total_tarefas; i++)
    {
        printf("Tarefa %d: %s | Prevista: dia %d | %s\n",
            tarefas[i].id_tarefa,
            tarefas[i].descricao,
            tarefas[i].data_prevista,
            tarefas[i].concluida ? "Concluida" : "Pendente");
    }
}

int tarefas_obter_por_indice(int indice, Tarefa *destino)
{
    if (indice < 0 || indice >= total_tarefas)
    {
        return 0;
    }

    *destino = tarefas[indice];
    return 1;
}

int tarefas_total(void)
{
    return total_tarefas;
}

```

## 5.8 include/io.h

```
/* io.h - funcoes de input/output */

#ifndef IO_H
#define IO_H

// mudei de scanf para fgets (erro D1)
void io_ler_string(const char *prompt, char *destino, int tamanho_max);

int io_ler_inteiro(const char *prompt);

// nunca usar fflush(stdin), uso getchar (erro D2)
void io_limpar_buffer(void);

void io_mostrar_menu(int data_atual);
void io_pausa(void);
int io_eof(void);

#endif /* IO_H */
```

## 5.9 src/io.c

```
/* =====
 * io.c – Implementação do módulo de Input/Output
 * GreenTrack – Gestão de Jardins Comunitários
 *
 * Corrigi aqui dois erros de input/output que eram bastante
 * problemáticos no código original:
 *   D1 – O scanf("%s") só lia até ao primeiro espaço, pelo que
 *   nomes como "Rosa Chinesa" ficavam truncados em "Rosa".
 *   Mudei para fgets, que lê a linha inteira.
 *   D2 – Após um scanf("%d"), o \n fica no buffer e a próxima
 *   leitura com fgets consumia esse \n e retornava vazio.
 *   O código original usava fflush(stdin), que é comportamento
 *   indefinido em C. Mudei para while(getchar() != '\n').
 *   Adicionei também deteção de EOF para evitar loops infinitos
 *   quando o programa é executado em modo pipe.
 * ===== */

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

/* Corrigi esta função para resolver o erro D1 – mudei de
 * scanf("%s") para fgets porque o scanf não lê strings com
 * espaços. Também trato o caso em que o input é maior que
 * o buffer (limpo o resto com io_limpar_buffer). */
```

```

void io_ler_string(const char *prompt, char *destino, int tamanho_max)
{
    int len;

    printf("%s", prompt);
    fflush(stdout);

    if (fgets(destino, tamanho_max, stdin) == NULL)
    {
        /* EOF atingido – marcar string como vazia */
        destino[0] = '\0';
        return;
    }

    len = strlen(destino);
    if (len > 0 && destino[len - 1] == '\n')
    {
        destino[len - 1] = '\0';
    }
    else if (len > 0)
    {
        /* Input maior que o buffer – limpar o resto */
        io_limpar_buffer();
    }
}

int io_ler_inteiro(const char *prompt)
{
    int valor;
    int resultado;

    printf("%s", prompt);
    fflush(stdout);

    resultado = scanf("%d", &valor);

    if (resultado == EOF)
    {
        return -9999; /* Código especial para EOF */
    }

    /* Correção D2: limpar buffer após leitura numérica – se não
       fizer isto, o próximo fgets come o \n e retorna vazio.
       Já tentei usar fflush(stdin) mas isso é indefinido em C. */
    io_limpar_buffer();

    if (resultado != 1)
    {

```



```

        return -1;
    }

    return valor;
}

/* Nunca usar fflush(stdin) – é comportamento indefinido segundo
   o standard C. Este ciclo descarta tudo até ao newline. */
void io_limpar_buffer(void)
{
    int c;

    while ((c = getchar()) != '\n' && c != EOF)
    {
        /* descartar caracter */
    }
}

int io_eof(void)
{
    return feof(stdin);
}

void io_mostrar_menu(int data_atual)
{
    printf("\n==== GreenTrack - Jardins Comunitarios =====\n");
    printf("  Data atual: dia %d\n", data_atual);
    printf("===== \n");
    printf("  1. Listar plantas\n");
    printf("  2. Adicionar planta\n");
    printf("  3. Registrar rega\n");
    printf("  4. Verificar regas necessarias\n");
    printf("  5. Listar tarefas pendentes\n");
    printf("  6. Criar tarefa\n");
    printf("  7. Concluir tarefa\n");
    printf("  8. Avancar dia\n");
    printf("  0. Sair\n");
    printf("===== \n");
    printf("Opcao: ");
}

void io_pausa(void)
{
    int c;

    printf("\nPrima Enter para continuar...");
    fflush(stdout);
}

```

```

while ((c = getchar()) != '\n' && c != EOF)
{
    /* descartar */
}
}

```

## 5.10 src/main.c

```

/* =====
 * main.c – Ponto de entrada do GreenTrack
 * GreenTrack – Gestão de Jardins Comunitários
 *
 * Menu interativo com do-while + switch-case.
 * Carrega dados no arranque; persistência imediata em cada
 * operação de escrita (conforme enunciado).
 *
 * Notas sobre correções aplicadas aqui:
 * - D1: Uso io_ler_string() em vez de scanf("%s") para ler
 *   strings com espaços (nomes, descrições).
 * - F1: Os IDs das plantas são gerados automaticamente dentro
 *   da função plantas_adicionar(), não são pedidos ao utilizador.
 * - G1+E1: A validação da planta e a atualização de ultima_rega
 *   são feitas dentro de regas_registar(), não no main.
 * - G2: plantas_verificar_rega() agora retorna int (quantas
 *   precisam de rega), embora aqui só imprima o resultado.
 * - B1/B2: As comparações com == foram corrigidas nos módulos
 *   respetivos (tarefas), não aqui no main.
 * ===== */

#include "tipos.h"
#include "plantas.h"
#include "regas.h"
#include "tarefas.h"
#include "io.h"

int main(void)
{
    int opcao;
    int data_atual = 0; /* dia 0 = 01/01/2026 */

    /* — Carregar dados persistidos ————— */
    plantas_inicializar();
    regas_inicializar();
    tarefas_inicializar();

    plantas_carregar("plantas.csv");
    regas_carregar("regas.csv");
    tarefas_carregar("tarefas.csv");

```

```

printf("Dados carregados: %d plantas, %d regas, %d tarefas.\n",
      plantas_total(), regas_total(), tarefas_total());

/* — Ciclo principal do menu ————— */
do
{
    io_mostrar_menu(data_atual);
    opcao = io_ler_inteiro("");

    /* Detecção de EOF — evita loop infinito em modo pipe */
    if (opcao == -9999)
    {
        printf("\nInput terminado. A guardar e sair...\n");
        plantas_guardar("plantas.csv");
        regas_guardar("regas.csv");
        tarefas_guardar("tarefas.csv");
        break;
    }

    switch (opcao)
    {
        case 1:
        {
            plantas_listar();
            io_pausa();
            break;
        }

        case 2:
        {
            /* Adicionar planta — uso io_ler_string para nomes
               com espaços (correção D1) */
            char nome[50], especie[50], data_plantio[11];
            int intervalo;

            printf("\n--- Adicionar Planta ---\n");

            io_ler_string("Nome: ", nome, sizeof(nome));
            io_ler_string("Especie: ", especie, sizeof(especie));
            io_ler_string("Data plantio (DD/MM/AAAA): ", data_plantio,
sizeof(data_plantio));
            intervalo = io_ler_inteiro("Intervalo de rega (dias): ");

            if (intervalo <= 0)
            {
                printf("Erro: intervalo de rega invalido.\n");
            }
        }
    }
}

```

```

else
{
    /* O ID é gerado automaticamente (correção F1) – não
       preciso de pedir ao utilizador */
    if (plantas_adicionar(nome, especie, data_plantio,
                          intervalo, data_atual))
    {
        printf("Planta adicionada com sucesso! (ID: %d)\n",
               plantas_total());
    }
}
io_pausa();
break;
}

case 3:
{
    /* Registrar rega – a validação da planta e a
       atualização de ultima_rega ficam dentro de
       regas_registar() (erros G1 e E1) */
    int id_planta, quantidade;

    printf("\n--- Registrar Rega ---\n");
    plantas_listar();

    id_planta = io_ler_inteiro("ID da planta: ");
    quantidade = io_ler_inteiro("Quantidade de agua (ml): ");

    if (quantidade <= 0)
    {
        printf("Erro: quantidade invalida.\n");
    }
    else
    {
        if (regas_registar(id_planta, data_atual, quantidade))
        {
            printf("Rega registada com sucesso!\n");
        }
    }
    io_pausa();
    break;
}

case 4:
{
    /* Verificar regas – a função agora retorna quantas
       precisam de rega (correção G2), mas aqui só
       imprimo o resultado */

```

```

        printf("\n--- Verificacao de Regas ---\n");
        plantas_verificar_rega(data_atual);
        io_pausa();
        break;
    }

    case 5:
    {
        /* Listar tarefas pendentes – a comparação
           concluida == 0 foi corrigida no módulo (erro B1) */
        printf("\n");
        tarefas_listar_pendentes();
        io_pausa();
        break;
    }

    case 6:
    {
        /* Criar tarefa – io_ler_string para descrições
           com espaços (D1), ID automático (F2) */
        char descricao[100];
        int data_prevista;

        printf("\n--- Criar Tarefa ---\n");

        io_ler_string("Descricao: ", descricao, sizeof(descricao));
        data_prevista = io_ler_inteiro("Data prevista (dia): ");

        if (data_prevista < 0)
        {
            printf("Erro: data invalida.\n");
        }
        else
        {
            if (tarefas_criar(descricao, data_prevista))
            {
                printf("Tarefa criada com sucesso!\n");
            }
        }
        io_pausa();
        break;
    }

    case 7:
    {
        /* Concluir tarefa – o == em vez de = já foi
           corrigido no módulo tarefas (erro B2) */
        int id_tarefa;

```

```

        printf("\n--- Concluir Tarefa ---\n");
        tarefas_listar_pendentes();

        id_tarefa = io_ler_inteiro("ID da tarefa a concluir: ");

        if (tarefas_concluir(id_tarefa))
        {
            printf("Tarefa concluida com sucesso!\n");
        }
        io_pausa();
        break;
    }

    case 8:
    {
        data_atual++;
        printf("Data avancada para o dia %d.\n", data_atual);
        io_pausa();
        break;
    }

    case 0:
    {
        printf("\nA guardar dados e a sair...\n");
        /* A persistência já é imediata em cada operação,
           mas guardo tudo antes de sair por segurança */
        plantas_guardar("plantas.csv");
        regas_guardar("regas.csv");
        tarefas_guardar("tarefas.csv");
        printf("Dados guardados com sucesso. Ate logo!\n");
        break;
    }

    default:
    {
        printf("Opcao invalida. Tente novamente.\n");
        io_pausa();
        break;
    }
}

} while (opcao != 0);

return 0;
}

```

## 5.11 Makefile

```
# Makefile simples para GreenTrack
# Compilacao modular com GCC

CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -I./include
OBJS = main.o plantas.o regas.o tarefas.o io.o
TARGET = greentrack.exe

$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $(OBJS)

main.o: src/main.c
    $(CC) $(CFLAGS) -c $< -o $@

plantas.o: src/plantas.c
    $(CC) $(CFLAGS) -c $< -o $@

regas.o: src/regas.c
    $(CC) $(CFLAGS) -c $< -o $@

tarefas.o: src/tarefas.c
    $(CC) $(CFLAGS) -c $< -o $@

io.o: src/io.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJS) $(TARGET)

.PHONY: clean
```