



**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

## TRABALHO / RESOLUÇÃO:

<b>Introdução .....</b>	<b>3</b>
<b>Tarefa 1 - Melhorias de Legibilidade e Documentação .....</b>	<b>3</b>
<b>Tarefa 2 - Testes de Unidade.....</b>	<b>7</b>
<b>Tarefa 3 - Testes de Integração.....</b>	<b>10</b>
<b>Conclusão .....</b>	<b>13</b>
<b>Anexos.....</b>	<b>14</b>
<b>Evidências de execução - Testes de Execução .....</b>	<b>14</b>
<b>Evidências de execução - Testes de Integração .....</b>	<b>15</b>
<b>Estrutura do Código Entregue .....</b>	<b>15</b>
<b>Makefile Endpoints.....</b>	<b>16</b>
<b>Ficheiro: Makefile .....</b>	<b>16</b>
<b>Ficheiro: include/tipos.h .....</b>	<b>18</b>
<b>Ficheiro: include/plantas.h .....</b>	<b>20</b>
<b>Ficheiro: include/regas.h .....</b>	<b>23</b>
<b>Ficheiro: include/tarefas.h .....</b>	<b>25</b>
<b>Ficheiro: include/io.h .....</b>	<b>28</b>
<b>Ficheiro: src/main.c.....</b>	<b>30</b>
<b>Ficheiro: src/plantas.c .....</b>	<b>34</b>
<b>Ficheiro: src/regas.c .....</b>	<b>40</b>
<b>Ficheiro: src/tarefas.c .....</b>	<b>44</b>
<b>Ficheiro: src/io.c.....</b>	<b>48</b>
<b>Ficheiro: testes/testes_unidade.c .....</b>	<b>51</b>
<b>Ficheiro: testes/testes_integracao.c.....</b>	<b>64</b>

## INTRODUÇÃO

Neste relatório, documento o trabalho que desenvolvi no âmbito do E-fólio B, assente no código base do E-fólio A ( sistema GreenTrack - Gestão de Jardins Comunitários ), que respeitava os princípios de modularidade do Módulo 2 (separação .h/.c, encapsulamento com static, include guards, responsabilidade única).

O objetivo do E-fólio B é duplo:

1º - elevar a legibilidade e a documentação do código para o padrão profissional exigido pelo Módulo 3 (RF 3.3),

2º - implementar testes sistemáticos de unidade e de integração conforme o Módulo 4 (RF 4.2 e RF 4.3).

A metodologia que adotei foi incremental: em vez de reescrever código que já estava correto, concentrei-me nas fragilidades identificadas pelos testes e na documentação para contratos formais Doxygen.

## TAREFA 1 - MELHORIAS DE LEGIBILIDADE E DOCUMENTAÇÃO

### Estratégia Adotada

O código base do E-fólio A já possuía uma arquitetura modular: funções pequenas ( $\leq 30$  linhas), guard clauses em todas as operações de escrita, nomenclatura consistente e expressiva (ex: padrão módulo\_verbo()), e persistência imediata em CSV.

Forçar alterações estruturais onde não existiam problemas reais violaria o princípio de clareza do RF 3.3 (secção 7).

Assim, a minha estratégia concentrou-se em três pontos:

- 1. Documentação Doxygen nas interfaces públicas (.h):**  
Substitui os comentários informais e as notas históricas de erros por contratos formais com @brief, @param, @return, @pre e @post.
- 2. Transição de comentários de "histórico de bugs" para "justificação de arquitetura":**  
O código do E-fólio A continha extensas narrativas na primeira pessoa (corrigi, tentei, percebi) que descreviam o processo de debug. Substitui-as por comentários impessoais que explicam as decisões de design (o porquê da arquitetura).
- 3. Remoção de código morto:**  
Eliminei protótipos comentados de versões antigas, blocos de debug, e comentários redundantes sobre código autoexplicativo.

## Exemplo Ilustrativo: Antes vs. Depois

### Antes (E-fólio A - 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
```

### Depois (E-fólio B - plantas.h):

```
/**
 * @brief Adiciona uma nova planta ao array interno.
 *
 * O ID da planta é gerado automaticamente (max_id + 1), eliminando
 * a necessidade de input do utilizador e prevenindo duplicações.
 *
 * @param nome      Nome comum da planta (máx. 49 caracteres + '\0').
 * @param especie   Nome científico ou espécie (máx. 49 + '\0').
 * @param data_plantio Data no formato DD/MM/AAAA (máx. 10 + '\0').
 * @param intervalo  Número de dias entre regas (deve ser > 0).
 * @param data_atual Dia atual do sistema (timestamp: dias desde
 *                  01/01/2026), usado para inicializar ultima_rega.
 * @return 1 se a planta foi adicionada e persistida com sucesso;
 *         0 se o array atingiu MAX_PLANTAS ou se os parâmetros são
 *         inválidos (ex: intervalo <= 0).
 * @pre  O módulo deve estar inicializado.
 * @post O array contém a nova planta no final; total_plantas
 *       incrementado; ficheiro CSV sincronizado.
 */
int plantas_adicionar(const char *nome, const char *especie,
                     const char *data_plantio, int intervalo,
                     int data_atual);
```

**Análise da melhoria:** A documentação Doxygen transforma o ficheiro de cabeçalho num contrato formal. Um programador que reutilize este módulo sabe, sem ler o .c, quais são as pré-condições,

as pós-condições e os limites dos parâmetros. Isto é essencial para a reutilização e para a escrita de testes derivados do contrato.

### Alternativa Rejeitada: Centralizar Validações no main.c

Outra opção que ponderuei foi centralizar todas as validações de segurança no main.c, antes de invocar as funções dos módulos. Por exemplo, no menu de regas, poderia verificar se a planta existe e se a quantidade de água é positiva antes de chamar regas\_registar().

No entanto, rejeitei esta abordagem porque violava o princípio de encapsulamento e responsabilidade única: a lógica defensiva ficaria fragmentada no ponto de invocação, em vez de residir junto aos dados que protege. Se outro ponto do sistema (por exemplo, um futuro módulo de automação) invocasse regas\_registar() diretamente, a validação estaria ausente. Assim, decidi manter as verificações dentro do próprio módulo regas, garantindo que cada unidade protege a sua própria integridade independentemente do contexto de chamada.

## 2.º Exemplo: Refatoração do Módulo de I/O (io.h)

Antes (E-fólio A - 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);
```

Depois (E-fólio B - io.h):

```
/**
 * @file io.h
 * @brief Interface pública do módulo de Input/Output (GreenTrack).
 *
 * Este módulo centraliza todas as operações de interação com o
 * utilizador e com o sistema de ficheiros, garantindo leituras
 * robustas, limpeza de buffers e deteção de EOF.
 *
 * @author 2403882
 * @version E-fólio B
 * @date 2026
 */

#ifndef IO_H
#define IO_H
```

```

/**
 * @brief Lê uma string do teclado com limite de tamanho.
 *
 * Utiliza fgets() em vez de scanf("%s") para preservar espaços
 * internos na string. Remove automaticamente o caractere newline
 * e esvazia o buffer se o input exceder o tamanho máximo.
 *
 * @param[in]  prompt      Mensagem a apresentar ao utilizador.
 * @param[out] destino     Buffer onde a string lida será escrita.
 * @param[in]  tamanho_max Capacidade do buffer (incluindo '\0').
 * @pre destino != NULL && tamanho_max > 0.
 * @post destino contém a string lida (truncada se necessário);
 *       o buffer stdin fica vazio e pronto para a próxima leitura.
 */
void io_ler_string(const char *prompt, char *destino, int tamanho_max);

```

**Análise da melhoria:** Esta refatoração transversal foi essencial para prevenir falhas de leitura que observámos no E-fólio A, nomes compostos como "Rosa Chinensis" ficavam truncados por `scanf("%s")` e o buffer residual de `scanf("%d")` consumia a leitura seguinte. O cabeçalho original descrevia o processo de descoberta com apontamentos informais ("mudei de `scanf` para `fgets`", "erro D1", "nunca usar `fflush(stdin)`"). Substituí-o por contratos Doxygen que explicam porquê `fgets` em vez de `scanf` e que documentam as pré-condições e pós-condições da limpeza do buffer. Desta forma, quem reutiliza o módulo compreende o estado de `stdin` sem ter de ler a implementação, um passo decisivo para a reutilização profissional.

### Justificação das Partes Mantidas Inalteradas

Conforme a nota do enunciado, "não se pretende que reescrevas o teu código, mas que o melhores." mantive as seguintes características idênticas:

- **Funções pequenas:** Nenhuma função pública excede 30 linhas; não vi necessidade de extração artificial.
- **Guard clauses:** Já aplicadas em todas as funções de escrita; adicioná-las onde já existem seria redundante.
- **Nomenclatura:** Já em padrão `modulo_verbo()`.
- **Estruturas de dados:** Mantive `MAX_PLANTAS = 100`, `MAX_REGAS = 500`, `MAX_TAREFAS = 200`; foquei a melhoria na justificação dos valores, não na alteração.
- **Lógica de negócio:** Preservei o cálculo de `ultima_rega`, o estado binário `concluida`  $\in \{0,1\}$ , e o fluxo do menu.

## TAREFA 2 - TESTES DE UNIDADE

### Funções e Cenários Testados

Foram selecionadas 5 funções críticas, distribuídas pelos 3 módulos principais, totalizando 15 cenários:

#### Demonstração do Padrão de Diagnóstico

Para demonstrar a aplicação prática do padrão exigido pela alínea 2.c, apresento o seguinte excerto do meu código de teste de unidade, que ilustra o diagnóstico com `printf()` para valores obtidos versus esperados e `assert()` para paragem automática em caso de falha:

```
void teste_plantas_adicionar_normal(void)
{
    int obtido, esperado;
    Planta p;

    printf("TESTE: plantas_adicionar() [Normal] – adicionar planta valida\n");

    plantas_inicializar();

    obtido = plantas_adicionar("Rosa", "Rosa chinensis", "15/03/2026", 7, 0);
    esperado = 1;
    if (obtido != esperado)
        printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
    assert(obtido == esperado);

    /* Verificar pós-condição: total == 1 */
    if (plantas_total() != 1)
        printf(" FALHA: total esperado 1, obtido %d\n", plantas_total());
    assert(plantas_total() == 1);

    /* Verificar pós-condição: ID atribuído automaticamente */
    if (plantas_obter_por_indice(0, &p) != 1)
        printf(" FALHA: nao conseguiu obter planta no indice 0\n");
    assert(plantas_obter_por_indice(0, &p) == 1);

    if (p.id != 1)
        printf(" FALHA: id esperado 1, obtido %d\n", p.id);
    assert(p.id == 1);

    /* Verificar pós-condição: ultima_rega == data_atual (0) */
    if (p.ultima_rega != 0)
        printf(" FALHA: ultima_rega esperado 0, obtido %d\n", p.ultima_rega);
    assert(p.ultima_rega == 0);

    printf(" OK\n");
}
```

Função	Cenário	Tipo	Objectivo	Resultado
plantas_adicionar()	Adicionar planta válida	[Normal]	Retorno 1, total=1, ID=1, ultima_rega=0	PASSOU
plantas_adicionar()	Encher até MAX=100	[Limite]	100ª OK, 101ª falha, total preservado	PASSOU
plantas_adicionar()	Intervalo inválido (0)	[Erro]	Retorno 0, estado preservado	PASSOU*
plantas_obter_por_id()	Planta existente	[Normal]	Índice $\geq 0$ , nome correto	PASSOU
plantas_obter_por_id()	Planta inexistente	[Erro]	Retorno -1	PASSOU
plantas_obter_por_id()	Procurar ID máximo após encher array	[Limite]	Retorno $\geq 0$ , ID=MAX_PLANTAS	PASSOU
regas_registar()	Rega válida	[Normal]	Retorno 1, total=1, ultima_rega atualizada	PASSOU
regas_registar()	Encher até MAX_REGAS (500)	[Limite]	500ª OK, 501ª falha	PASSOU
tarefas_criar()	Data prevista inválida (-1)	[Erro]	Retorno 0, estado preservado	PASSOU
tarefas_concluir()	Concluir na última posição	[Limite]	Retorno 1, concluida=1	PASSOU
regas_registar()	Planta inexistente	[Erro]	Retorno 0, estado preservado	PASSOU
tarefas_criar()	Tarefa válida	[Normal]	Retorno 1, concluida=0, descrição correta	PASSOU
tarefas_criar()	Encher até MAX=200	[Limite]	200ª OK, 201ª falha, total preservado	PASSOU
tarefas_concluir()	Concluir existente	[Normal]	Retorno 1, concluida=1	PASSOU
tarefas_concluir()	Inexistente	[Erro]	Retorno 0, estado preservado	PASSOU

\* Falhou inicialmente; correção aplicada.



## Análise do Bug Detetado: intervalo <= 0

Durante a primeira execução de testes\_unidade.exe, o teste [Erro] - intervalo inválido, falhou com um Assertion failed e provocou a paragem imediata do programa:

```
user@DESKTOP-DHDDM05:~/labprog$ cd codigo_efoliob && make clean && make testes && ./testes_unidade.exe
rm -rf obj greentrack.exe testes_unidade.exe testes_integracao.exe
mkdir -p obj
gcc -Wall -Wextra -std=c99 -I./include -c src/plantas.c -o obj/plantas.o
gcc -Wall -Wextra -std=c99 -I./include -c src/regas.c -o obj/regas.o
gcc -Wall -Wextra -std=c99 -I./include -c src/tarefas.c -o obj/tarefas.o
gcc -Wall -Wextra -std=c99 -I./include -c src/io.c -o obj/io.o
gcc -Wall -Wextra -std=c99 -I./include -c testes/testes_unidade.c -o obj/testes_unidade.o
gcc -Wall -Wextra -std=c99 -I./include -o testes_unidade.exe obj/plantas.o obj/regas.o obj/tarefas.o obj/io.o obj/testes_unidade.o
=====
Testes de Unidade - GreenTrack
Padrão: printf() + assert()
=====
TESTE: plantas_adicionar() [Normal] - adicionar planta valida
OK
TESTE: plantas_adicionar() [Limite] - encher array ate MAX_PLANTAS (100)
Erro: limite de plantas atingido (100).
OK
TESTE: plantas_adicionar() [Erro] - intervalo invalido (<= 0)
FALHA: retorno esperado 0, obtido 1
testes_unidade.exe: testes/testes_unidade.c:134: teste_plantas_adicionar_erro_intervalo: Assertion 'obtido == esperado' failed.
Aborted (core dumped)
user@DESKTOP-DHDDM05:~/labprog/codigo_efoliob$
```

**Diagnóstico:** A função plantas\_adicionar() não validava o parâmetro intervalo. Embora o main.c verificasse intervalo <= 0 antes de invocar a função, o módulo não possuía uma guard clause defensiva.

**Correção:**

```
/* Guard clause: intervalo deve ser estritamente positivo */
if (intervalo <= 0)
{
    printf("Erro: intervalo de rega invalido (%d). Deve ser > 0.\n",
intervalo);
    return 0;
}
```

**Impacto:** A correção fortaleceu o contrato defensivo do módulo sem alterar o comportamento observável pelo utilizador final.

Tal como referi no E-fólio A, sentia que a solução base era muito robusta, os 18 erros estavam corrigidos e a arquitetura modular parecia-me sólida.

No entanto, a implementação rigorosa dos testes de unidade revelou-me uma falha cega que me tinha escapado completamente: o módulo plantas confiava cegamente na validação feita pelo main.c, sem uma guard clause defensiva própria.

Foi uma lição prática de que herdar código funcional ( mesmo código próprio e bem avaliado ) não significa herdar código blindado. Isto mostrou-me que os testes de unidade não são meramente verificativos, são instrumentos de descoberta de fragilidades latentes.

## Análise de Sensibilidade N±1

Segundo o RF 4.3, "os valores de teste devem exercitar as fronteiras do domínio." os testes de limite exploram explicitamente N±1:

### **Módulo plantas - MAX=100:**

- Preenchemos 99 plantas (N-1).
- A 100ª é aceite (N → retorno 1, total=100).
- A 101ª é rejeitada (N+1 → retorno 0, total=100).

### **Módulo tarefas - MAX=200:**

- Preenchemos 199 tarefas (N-1).
- A 200ª é aceite (N → retorno 1, total=200).
- A 201ª é rejeitada (N+1 → retorno 0, total=200).

### **Porque N±1 é crítico em C:**

Arrays estáticos não possuem verificação automática de limites; um acesso a `array[MAX]` resulta em comportamento indefinido. O compilador GCC com `-Wall -Wextra` não deteta este erro para acessos dependentes de variáveis. O teste aos limites  $N \pm 1$  demonstra que a guard clause `total >= MAX` está posicionada corretamente antes do acesso, prevenindo a corrupção de memória.

## **TAREFA 3 - TESTES DE INTEGRAÇÃO**

### **A insuficiência dos testes de unidade**

Os testes de unidade verificam contratos internos de cada função isoladamente. No entanto, num sistema modular, o comportamento correto de cada função individual não garante a cooperação correta entre módulos. Considerei três exemplos concretos do GreenTrack:

#### **Exemplo 1 - `regas_registar()` e `plantas_atualizar_ultima_rega()`**

Um teste de unidade de `regas_registar()` poderia substituir a chamada a `plantas_atualizar_ultima_rega()` por um stub. Nesse caso, o teste de unidade passaria, mas um bug no stub (ex: atualizar campo errado da planta) permaneceria oculto. Apenas o teste de integração revela a falha porque exercita a cadeia real de chamadas.

#### **Exemplo 2 - `tarefas_concluir()` e persistência**

Um teste de unidade verifica `concluida == 1` em memória. Se `tarefas_guardar()` fosse acidentalmente removida de `tarefas_concluir()`, o teste de unidade continuaria a passar. O teste de integração falharia no passo de recarregar, porque o CSV não conteria a atualização.

#### **Exemplo 3 - CSV corrompido**

Um teste de unidade com dados mockados poderia simular sucesso, mas o cenário de corrupção real (dados externos malformados) só é significativo quando testado com ficheiros reais do disco.

### Cenários de Integração Testados

ID	Cenário	Módulos	Objectivo	Resultado
A	Cooperação regas→plantas	regas, plantas	Verificar se regas_registar() atualiza ultima_rega da planta corretamente	PASSOU
B	Persistência tarefas	tarefas, I/O	Ciclo: concluir → guardar CSV → reiniciar → carregar → verificar estado	PASSOU
C1	CSV inexistente	plantas, I/O	Robustez: retorno 0 sem crash, estado vazio preservado	PASSOU
C2	CSV corrompido	plantas, I/O	Parar na linha malformada (3 campos em vez de 6), preservar válidos	PASSOU

#### Cenário A -

**Fluxo:** Adicionar planta "Rosa" → registrar rega (dia 5, 500ml) → verificar: total\_regas=1, planta.ultima\_rega=5, rega.id\_planta=1, rega.quantidade=500.

**Resultado Esperado:** O campo ultima\_rega deve ser atualizado para o valor correto (5).

**Resultado Obtido:** O estado em memória confirmou a alteração exata do campo ultima\_rega e o incremento do contador total\_regas.

#### Cenário B -

**Fluxo:** Criar tarefa "Regar rosas" → verificar concluida=0 → concluir → verificar concluida=1 em memória → simular reinício (inicializar + carregar) → verificar concluida=1 recuperada do CSV.

**Resultado Esperado:** Após simular reinício, o estado concluida=1 deve ser recuperado do CSV.

**Resultado Obtido:** O estado persistido foi corretamente carregado e a tarefa apresentou concluida=1.

#### Cenário C1 -

**Justificação:** Segundo o RF 4.2, "os programas devem reagir de forma previsível a entradas inesperadas." Um ficheiro inexistente na primeira execução é um cenário normal, não um erro. O módulo deve continuar com array vazio.

**Resultado Esperado:** plantas\_carregar() deve devolver 0 e o programa continuar com o array vazio.

**Resultado Obtido:** A função devolveu 0, o array permaneceu vazio e o programa prosseguiu sem terminação abrupta.

## **Cenário C2 -**

**Robustez:** Criámos um CSV com linha válida + linha malformada (só 3 campos em vez de 6). O parser para corretamente na linha malformada (fscanf retorna 3 campos, espera 6 → break), preservando o registo válido anterior.

**Resultado Esperado:** A linha malformada deve ser ignorada e os registos válidos anteriores preservados.

**Resultado Obtido:** O parser interrompeu o processamento na linha inválida, preservou os registos anteriores e o programa continuou a execução de forma segura com os dados íntegros.

## **Reflexão Obrigatória (Alíneas c e d)**

### **Alínea c) - Função difícil de testar de forma unitária:**

Identifico a função `plantas_carregar()` como particularmente difícil de isolar num teste de unidade puro. A dificuldade advém da sua dependência simultânea de dois fatores externos ao escopo da função: o estado global do módulo (o array static de plantas e o contador `total_plantas`, inacessíveis de fora) e o sistema de ficheiros (a função abre e lê diretamente um ficheiro CSV do disco através de `fopen()` e `fscanf()`). Para testá-la de forma rigorosa, sou forçado a criar ficheiros temporários no sistema de ficheiros e a confiar na manipulação do estado interno, o que quebra o princípio de isolamento próprio dos testes de unidade. Se fosse a redesenhar esta função, optaria por uma mudança arquitetural de injeção de dependências: passaria o array de plantas e o seu tamanho como parâmetros de entrada/saída, e receberia um `FILE*` já aberto em vez de o nome do ficheiro. Desta forma, o teste unitário poderia alimentar a função com um descritor de ficheiro temporário em memória e verificar o estado do array diretamente, sem tocar no sistema de ficheiros real.

### **Alínea d) - Reação do sistema a ficheiro CSV corrompido ou inexistente:**

A robustez do sistema face a dados externos malformados é demonstrada pelos testes de integração C1 e C2. Quando o programa inicia e o ficheiro `plantas.csv` ainda não existe (cenário C1), a função `plantas_carregar()` devolve 0, imprime uma mensagem informativa e permite que o programa continue a executar com o array vazio, não há terminação abrupta. No cenário C2, quando o ficheiro contém linhas malformadas (por exemplo, apenas 3 campos em vez dos 6 esperados), o ciclo de leitura valida o retorno de `fscanf()`: se o número de campos lidos for diferente do esperado, a função interrompe o processamento daquele registo com `break`, ignorando a linha inválida e preservando os registos válidos já carregados anteriormente. O programa continua a sua execução de forma segura, com os dados íntegros que conseguiu extrair. Esta abordagem defensiva assegura que um ficheiro parcialmente corrompido não compromete a totalidade da base de dados.

## CONCLUSÃO

O E-fólio B demonstrou que a legibilidade e os testes sistemáticos são complementares, não substituíveis. A documentação Doxygen com contratos formais (@pre, @post) não apenas eleva a qualidade do código, mas também serve de especificação para os testes: cada assert() valida uma pós-condição documentada, e cada cenário de erro valida uma pré-condição violada.

A falha inicial em plantas\_adicionar(intervalo=0) prova que os testes são instrumentos de descoberta, não mera verificação. O bug existia latente no código base ( no E-fólio A) porque nunca tinha sido exercitado por um teste que violasse a pré-condição.

A correção fortaleceu o contrato defensivo do módulo, alinhando-se com o princípio de programação defensiva do RF 4.3.

Os testes de integração, por sua vez, validaram fluxos que atravessam fronteiras de módulos. Enquanto a unidade verifica contratos internos, a integração verifica que os contratos são respeitados na cooperação, o que é essencial num sistema modular onde o estado de um módulo afeta o comportamento de outro.

Refletindo sobre o percurso desta unidade curricular, o Módulo 2 dá a base arquitetural, onde aprendi a separar interfaces de implementação e a respeitar o encapsulamento. Mas foram os Módulos 3 e 4 que verdadeiramente transformaram a forma de encarar código legado: percebi que a legibilidade profissional não é um "extra estético", mas uma ferramenta de sobrevivência quando se lê código de outra pessoa (ou do próprio eu do passado); e que os testes sistemáticos são a única forma de "dormir descansado" quando se altera uma linha num sistema com dependências cruzadas.

Se fosse a continuar a desenvolver o GreenTrack, consideraria alocação dinâmica, um formato de ficheiro mais robusto e, acima de tudo, expandir o conjunto de testes.

## ANEXOS

### Evidências de execução - Testes de Execução

```
user@DESKTOP-DHDDM05:~/labprog$ cd codigo_efoliob && make clean && make testes && ./testes_unidade
rm -rf obj greentrack greentrack.exe testes_unidade testes_integracao
mkdir -p obj
gcc -Wall -Wextra -std=c99 -I./include -c src/plantas.c -o obj/plantas.o
gcc -Wall -Wextra -std=c99 -I./include -c src/regas.c -o obj/regas.o
gcc -Wall -Wextra -std=c99 -I./include -c src/tarefas.c -o obj/tarefas.o
gcc -Wall -Wextra -std=c99 -I./include -c src/io.c -o obj/io.o
gcc -Wall -Wextra -std=c99 -I./include -c testes/testes_unidade.c -o obj/testes_unidade.o
gcc -Wall -Wextra -std=c99 -I./include -o testes_unidade obj/plantas.o obj/regas.o obj/tarefas.o obj/io.o obj/testes_unidade.o
=====
Testes de Unidade - GreenTrack
Padrão: printf() + assert()
=====

TESTE: plantas_adicionar() [Normal] - adicionar planta valida
OK
TESTE: plantas_adicionar() [Limite] - encher array ate MAX_PLANTAS (100)
Erro: limite de plantas atingido (100).
OK
TESTE: plantas_adicionar() [Erro] - intervalo invalido (<= 0)
Erro: intervalo de rega invalido (0). Deve ser > 0.
OK
TESTE: plantas_obter_por_id() [Normal] - procurar planta existente
OK
TESTE: plantas_obter_por_id() [Erro] - procurar planta inexistente
OK
TESTE: plantas_obter_por_id() [Limite] - procurar ID maximo apos encher array
OK
TESTE: regas_registar() [Normal] - regar planta existente
OK
TESTE: regas_registar() [Erro] - planta inexistente
Erro: planta com ID 999 nao existe.
OK
TESTE: regas_registar() [Limite] - encher array ate MAX_REGAS (500)
Erro: limite de regas atingido (500).
OK
TESTE: tarefas_criar() [Normal] - criar tarefa valida
OK
TESTE: tarefas_criar() [Limite] - encher array ate MAX_TAREFAS (200)
Erro: limite de tarefas atingido (200).
OK
TESTE: tarefas_criar() [Erro] - data prevista invalida (< 0)
Erro: data prevista invalida (-1). Deve ser >= 0.
OK
TESTE: tarefas_concluir() [Normal] - concluir tarefa existente
OK
TESTE: tarefas_concluir() [Erro] - tarefa inexistente
Erro: tarefa com ID 999 nao encontrada.
OK
TESTE: tarefas_concluir() [Limite] - concluir tarefa na ultima posicao (200)
OK

=====
Todos os testes passaram com sucesso!
=====
user@DESKTOP-DHDDM05:~/labprog/codigo_efoliob$
```

**Resultado:** 0 erros, 0 warnings na compilação; 15 cenários executados, 15 passaram

## Evidências de execução - Testes de Integração

```
user@DESKTOP-DHDDM05:~/labprog$ cd codigo_efoliob && make clean && make integracao && ./testes_integracao
rm -rf obj greentrack greentrack.exe testes_unidade testes_integracao
mkdir -p obj
gcc -Wall -Wextra -std=c99 -I./include -c src/plantas.c -o obj/plantas.o
gcc -Wall -Wextra -std=c99 -I./include -c src/regas.c -o obj/regas.o
gcc -Wall -Wextra -std=c99 -I./include -c src/tarefas.c -o obj/tarefas.o
gcc -Wall -Wextra -std=c99 -I./include -c src/io.c -o obj/io.o
gcc -Wall -Wextra -std=c99 -I./include -c testes/testes_integracao.c -o obj/testes_integracao.o
gcc -Wall -Wextra -std=c99 -I./include -o testes_integracao obj/plantas.o obj/regas.o obj/tarefas.o obj/
io.o obj/testes_integracao.o
=====
Testes de Integracao - GreenTrack
Padrão: printf() + assert()
=====

TESTE [Integracao A]: regas_registar() → atualiza ultima_rega em plantas
OK
TESTE [Integracao B]: tarefas_concluir() → persistencia em tarefas.csv
OK
TESTE [Integracao C1]: plantas_carregar() com CSV inexistente
OK
TESTE [Integracao C2]: plantas_carregar() com CSV corrompido
OK

=====
Todos os testes de integracao passaram!
=====
user@DESKTOP-DHDDM05:~/labprog/codigo_efoliob$
```

**Resultado:** 0 erros, 0 warnings na compilação; 4 cenários de integração executados, 4 passaram.

## Estrutura do Código Entregue

codigo\_efoliob/

- |— Makefile
- |— include/
  - | |— tipos.h
  - | |— plantas.h
  - | |— regas.h
  - | |— tarefas.h
  - | |— io.h
- |— src/
  - | |— main.c
  - | |— plantas.c
  - | |— regas.c
  - | |— tarefas.c
  - | |— io.c
- |— testes/
  - | |— testes\_unidade.c
  - | |— testes\_integracao.c
- |— plantas.csv
- |— regas.csv
- |— tarefas.csv

## Makefile Endpoints

Endpoint	Descrição
make all	Compila a aplicação principal nativamente (greentrack)
make testes	Compila os testes de unidade (testes_unidade)
make integracao	Compila os testes de integração (testes_integracao)
make mingw	Cross-compile de Linux para Windows (greentrack.exe)
make clean	Remove objetos e executáveis

## Ficheiro: Makefile

```
# =====
# Makefile – GreenTrack: Gestão de Jardins Comunitários
# Compilação modular com pastas include/ e src/
# =====

CC      = gcc
CFLAGS  = -Wall -Wextra -std=c99 -I./include
SRCDIR  = src
OBJDIR  = obj
TARGET  = greentrack

# Ficheiros objeto da aplicação principal
OBJS = $(OBJDIR)/plantas.o \
       $(OBJDIR)/regas.o \
       $(OBJDIR)/tarefas.o \
       $(OBJDIR)/io.o \
       $(OBJDIR)/main.o

# Objetos dos testes de unidade (sem main.o)
TEST_OBJS = $(OBJDIR)/plantas.o \
            $(OBJDIR)/regas.o \
            $(OBJDIR)/tarefas.o \
            $(OBJDIR)/io.o \
            $(OBJDIR)/testes_unidade.o
TEST_TARGET = testes_unidade

# Objetos dos testes de integração (sem main.o)
INTEG_OBJS = $(OBJDIR)/plantas.o \
            $(OBJDIR)/regas.o \
            $(OBJDIR)/tarefas.o \
            $(OBJDIR)/io.o \
            $(OBJDIR)/testes_integracao.o
INTEG_TARGET = testes_integracao

# Regra principal
all: $(OBJDIR) $(TARGET)
```



```

# Regra dos testes de unidade
testes: $(OBJDIR) $(TEST_TARGET)

# Regra dos testes de integração
integracao: $(OBJDIR) $(INTEG_TARGET)

# Criar pasta obj/
$(OBJDIR):
    mkdir -p $(OBJDIR)

# Link da aplicação principal
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $@ $^

# Link dos testes de unidade
$(TEST_TARGET): $(TEST_OBJS)
    $(CC) $(CFLAGS) -o $@ $^

# Link dos testes de integração
$(INTEG_TARGET): $(INTEG_OBJS)
    $(CC) $(CFLAGS) -o $@ $^

# Compilar ficheiros .c -> .o
$(OBJDIR)/%.o: $(SRCDIR)/%.c
    $(CC) $(CFLAGS) -c $< -o $@

# Compilar testes/testes_unidade.c -> obj/testes_unidade.o
$(OBJDIR)/testes_unidade.o: testes/testes_unidade.c
    $(CC) $(CFLAGS) -c $< -o $@

# Compilar testes/testes_integracao.c -> obj/testes_integracao.o
$(OBJDIR)/testes_integracao.o: testes/testes_integracao.c
    $(CC) $(CFLAGS) -c $< -o $@

# Cross-compilador para Windows (MinGW-w64)
MINGW_CC = x86_64-w64-mingw32-gcc

# Regra de cross-compilação para Windows
# Compila diretamente os ficheiros .c em greentrack.exe,
# sem ficheiros objeto intermediários, evitando conflitos
# com a compilação nativa Linux.
mingw: clean
    $(MINGW_CC) $(CFLAGS) -o $(TARGET).exe \
        $(SRCDIR)/plantas.c $(SRCDIR)/regas.c \
        $(SRCDIR)/tarefas.c $(SRCDIR)/io.c $(SRCDIR)/main.c

# Limpar
clean:
    rm -rf $(OBJDIR) $(TARGET) greentrack.exe $(TEST_TARGET) $(INTEG_TARGET)

```

```
.PHONY: all clean testes integracao mingw
```

### Ficheiro: include/tipos.h

```
/**
 * @file tipos.h
 * @brief Definições de tipos de dados e constantes do GreenTrack.
 *
 * Este ficheiro centraliza todas as estruturas de dados e limites
 * do sistema, garantindo consistência entre os diferentes módulos.
 * As constantes de limite foram definidas tendo em conta a
 * capacidade esperada de um jardim comunitário de média dimensão
 * e a memória disponível em sistemas embebidos.
 *
 * @author 2403882
 * @version E-fólio B
 * @date 2026
 */

#ifndef TIPOS_H
#define TIPOS_H

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

/* — Constantes de limite ————— */

/**
 * @brief Capacidade máxima do array de plantas.
 *
 * Valor de 100 plantas: suficiente para um jardim comunitário
 * de média dimensão sem comprometer a memória do sistema.
 */
#define MAX_PLANTAS 100

/**
 * @brief Capacidade máxima do array de regas.
 *
 * Valor de 500 regas: permite múltiplas regas diárias por
 * planta ao longo de vários meses de operação.
 */
```

```

#define MAX_REGAS    500

/**
 * @brief Capacidade máxima do array de tarefas.
 *
 * Valor de 200 tarefas: adequado para a gestão de atividades
 * semanais ao longo de um ano.
 */
#define MAX_TAREFAS  200

/* — Estruturas de dados — */

/**
 * @brief Representa uma planta no sistema GreenTrack.
 *
 * O campo @c id é atribuído automaticamente pelo módulo plantas.
 * O campo @c ultima_rega utiliza um timestamp relativo (dias desde
 * 01/01/2026) para simplificar cálculos de intervalo.
 */
typedef struct {
    int id;                /**< Identificador único automático. */
    char nome[50];         /**< Nome comum da planta. */
    char especie[50];      /**< Espécie botânica. */
    char data_plantio[11]; /**< Data no formato DD/MM/AAAA. */
    int intervalo_rega;    /**< Intervalo entre regas, em dias. */
    int ultima_rega;      /**< Timestamp: dias desde 01/01/2026. */
} Planta;

/**
 * @brief Representa uma rega efetuada a uma planta.
 *
 * Cada rega está associada a uma planta via @c id_planta,
 * garantindo integridade referencial com o módulo plantas.
 */
typedef struct {
    int id_rega;           /**< Identificador único da rega. */
    int id_planta;         /**< Referência à planta regada. */
    int data_rega;         /**< Timestamp: dias desde 01/01/2026. */
    int quantidade_agua;   /**< Volume de água em mililitros. */
} Rega;

/**
 * @brief Representa uma tarefa de manutenção do jardim.
 *
 * O campo @c concluida utiliza uma convenção binária:
 * 0 = pendente, 1 = concluída. Esta simplificação facilita
 * filtros e listagens por estado.
 */
typedef struct {
    int id_tarefa;        /**< Identificador único automático. */
    char descricao[100];  /**< Descrição textual da tarefa. */

```

```

    int data_prevista; /**< Timestamp: dias desde 01/01/2026. */
    int concluida;     /**< Estado: 0 = pendente, 1 = concluída. */
} Tarefa;

#endif /* TIPOS_H */

```

## Ficheiro: include/plantas.h

```

/**
 * @file plantas.h
 * @brief Interface pública do módulo de gestão de plantas.
 *
 * Este módulo encapsula todas as operações sobre o conjunto de plantas
 * registadas no sistema GreenTrack, incluindo criação, consulta,
 * atualização e persistência em ficheiro CSV.
 *
 * O estado interno (array de plantas, contadores e gerador de IDs) é
 * mantido como dados privados (static) em plantas.c. O acesso externo
 * faz-se exclusivamente através das funções declaradas nesta interface.
 *
 * A persistência é imediata: cada operação de escrita (adicionar,
 * atualizar) provoca a sincronização automática com o ficheiro CSV,
 * garantindo tolerância a falhas inesperadas do programa.
 *
 * @author 2403882
 * @version E-fólio B
 * @date 2026
 */

#ifndef PLANTAS_H
#define PLANTAS_H

#include "tipos.h"

/* =====
 * Operações de ciclo de vida do módulo
 * ===== */

/**
 * @brief Inicializa o módulo, colocando o array interno no estado vazio.
 *
 * @pre O módulo não deve ter sido previamente inicializado com dados
 *       carregados de ficheiro, salvo se o objetivo for explicitamente
 *       o reset do estado.
 * @post O contador de plantas é zero e o próximo ID é 1.
 */
void plantas_inicializar(void);

```

```

/**
 * @brief Carrega plantas de um ficheiro CSV para a memória.
 *
 * O formato esperado do ficheiro é:
 * id,nome,especie,data_plantio,intervalo_rega,ultima_rega
 *
 * O ciclo de leitura utiliza fscanff() diretamente em vez de feof(),
 * evitando a duplicação do último registo – problema clássico de
 * gestão de ficheiros em C (feof só sinaliza EOF após uma leitura
 * falhar).
 *
 * @param nome_ficheiro Caminho do ficheiro CSV a carregar.
 * @return 1 se o ficheiro foi aberto e processado com sucesso;
 *         0 se o ficheiro não existe (estado do array mantém-se vazio).
 * @pre O array interno deve estar inicializado (chamar plantas_inicializar
 *       antes, ou garantir que não existam dados residuais).
 * @post O array interno contém as plantas lidas; o próximo ID é
 *        recalculado como max(id_existente) + 1, prevenindo colisões
 *        se o CSV tiver IDs com lacunas.
 */
int plantas_carregar(const char *nome_ficheiro);

/**
 * @brief Guarda o estado atual das plantas num ficheiro CSV.
 *
 * @param nome_ficheiro Caminho do ficheiro CSV de destino.
 * @return 1 se a escrita foi bem-sucedida;
 *         0 em caso de erro de abertura/escrita do ficheiro.
 * @pre O array interno deve estar consistente.
 * @post O ficheiro contém todos os registos atuais do array.
 */
int plantas_guardar(const char *nome_ficheiro);

/* =====
 * Operações de manipulação de plantas
 * ===== */

/**
 * @brief Adiciona uma nova planta ao array interno.
 *
 * O ID da planta é gerado automaticamente (max_id + 1), eliminando
 * a necessidade de input do utilizador e prevenindo duplicações.
 *
 * @param nome Nome comum da planta (máx. 49 caracteres + '\0').
 * @param especie Nome científico ou espécie (máx. 49 + '\0').
 * @param data_plantio Data no formato DD/MM/AAAA (máx. 10 + '\0').
 * @param intervalo Número de dias entre regas (deve ser > 0).
 * @param data_atual Dia atual do sistema (timestamp: dias desde
 *                   01/01/2026), usado para inicializar ultima_rega.
 * @return 1 se a planta foi adicionada e persistida com sucesso;

```

```

*      0 se o array atingiu MAX_PLANTAS ou se os parâmetros são
*      inválidos (ex: intervalo <= 0).
* @pre  O módulo deve estar inicializado.
* @post O array contém a nova planta no final; total_plantas
*      incrementado; ficheiro CSV sincronizado.
*/
int plantas_adicionar(const char *nome, const char *especie,
                     const char *data_plantio, int intervalo,
                     int data_atual);

/**
* @brief Lista todas as plantas registadas na consola.
*
* Se não existirem plantas, apresenta uma mensagem informativa.
*/
void plantas_listar(void);

/* =====
* Operações de consulta
* ===== */

/**
* @brief Obtém uma cópia da planta num determinado índice do array.
*
* @param indice Posição no array interno (0-based).
* @param destino Ponteiro para estrutura Planta onde o resultado
*      será copiado.
* @return 1 se o índice é válido e a cópia foi efetuada;
*      0 se o índice está fora dos limites.
* @pre destino != NULL.
*/
int plantas_obter_por_indice(int indice, Planta *destino);

/**
* @brief Procura uma planta pelo seu identificador único.
*
* @param id Identificador da planta a procurar.
* @param destino Ponteiro para estrutura Planta onde o resultado
*      será copiado.
* @return Índice da planta no array (>= 0) se encontrada;
*      -1 se não existir planta com o ID indicado.
* @pre destino != NULL.
*/
int plantas_obter_por_id(int id, Planta *destino);

/* =====
* Operações de atualização
* ===== */

/**
* @brief Atualiza o campo ultima_rega de uma planta e persiste.

```

```

*
* Esta função é tipicamente invocada pelo módulo regas quando
* uma rega é registada para a planta identificada.
*
* @param id_planta Identificador da planta a atualizar.
* @param data_rega Dia da rega (timestamp).
* @return 1 se a planta foi encontrada e atualizada com sucesso;
*         0 se a planta com id_planta não existe.
* @post Se encontrada, o campo ultima_rega da planta é atualizado
*       e o ficheiro CSV sincronizado.
*/
int plantas_atualizar_ultima_rega(int id_planta, int data_rega);

/* =====
* Operações de diagnóstico
* ===== */

/**
* @brief Verifica quais as plantas que necessitam de rega no dia atual.
*
* Uma planta precisa de rega quando:
* data_atual - ultima_rega >= intervalo_rega
*
* @param data_atual Dia atual do sistema (timestamp).
* @return Número de plantas que precisam de rega.
*       Imprime na consola a lista detalhada.
*/
int plantas_verificar_rega(int data_atual);

/**
* @brief Devolve o número total de plantas registadas.
*
* @return Valor de total_plantas (entre 0 e MAX_PLANTAS).
*/
int plantas_total(void);

#endif /* PLANTAS_H */

```

## Ficheiro: include/regas.h

```

/**
* @file regas.h
* @brief Interface pública do módulo de Regas (GreenTrack).
*
* Este módulo gere o registo de regas efetuadas a cada planta,
* garantindo a integridade referencial com o módulo `plantas`
* e a persistência imediata em ficheiro CSV.
*
* @author 2403882
* @version E-fólio B

```

```

* @date 2026
*/

#ifndef REGAS_H
#define REGAS_H

#include "tipos.h"

/**
 * @brief Inicializa o módulo de regas.
 *
 * Esvazia o array interno e repõe o contador de IDs automáticos.
 * Deve ser invocado uma única vez antes de qualquer outra operação.
 *
 * @pre O módulo plantas deve ter sido inicializado previamente
 *       se se pretender validar existência de plantas.
 * @post total_regas == 0 && proximo_id_rega == 1.
 */
void regas_inicializar(void);

/**
 * @brief Carrega regas a partir de ficheiro CSV.
 *
 * O formato esperado é: id_rega,id_planta,data_rega,quantidade_agua
 * por linha. A leitura interrompe-se se atingir MAX_REGAS.
 *
 * @param[in] nome_ficheiro Caminho para o ficheiro CSV.
 * @return 1 se a leitura foi bem-sucedida (mesmo que vazio),
 *         0 se o ficheiro não existe.
 * @pre nome_ficheiro != NULL.
 * @post O array interno contém as regas lidas; o próximo ID automático
 *       é recalculado como max(id_existentes)+1.
 */
int regas_carregar(const char *nome_ficheiro);

/**
 * @brief Guarda todas as regas em ficheiro CSV.
 *
 * @param[in] nome_ficheiro Caminho para o ficheiro de destino.
 * @return 1 se a escrita foi bem-sucedida,
 *         0 se ocorreu erro ao abrir o ficheiro.
 * @pre nome_ficheiro != NULL.
 * @post O ficheiro contém todas as regas em formato CSV.
 */
int regas_guardar(const char *nome_ficheiro);

/**
 * @brief Regista uma nova rega para uma planta existente.
 *
 * Valida a existência da planta referenciada e o limite de capacidade
 * do array antes de inserir. Atualiza automaticamente o campo

```



```

* ultima_rega da planta correspondente e persiste imediatamente.
*
* @param[in] id_planta  Identificador da planta a regar.
* @param[in] data       Dia da rega (valor inteiro arbitrário).
* @param[in] quantidade Volume de água em mililitros (>0).
* @return 1 se a rega foi registada com sucesso,
*         0 se o array está cheio ou a planta não existe.
* @pre 0 módulo plantas deve estar carregado; id_planta deve referenciar
*      uma planta válida.
* @post A rega é adicionada ao array; ultima_rega da planta é atualizada;
*       ficheiro regas.csv é reescrito.
*/
int regas_registar(int id_planta, int data, int quantidade);

/**
* @brief Lista todas as regas no ecrã.
*
* Para cada rega, apresenta o ID da rega, o ID da planta, a data,
* a quantidade de água e, quando possível, o nome da planta.
*
* @pre 0 módulo plantas deve estar carregado para resolução de nomes.
* @post Nenhuma alteração de estado.
*/
void regas_listar(void);

/**
* @brief Obtém uma cópia da rega num dado índice.
*
* @param[in]  indice  Índice no array interno (0-based).
* @param[out] destino Estrutura onde a cópia será escrita.
* @return 1 se o índice é válido e a cópia foi efetuada,
*        0 se o índice está fora dos limites.
* @pre destino != NULL.
* @post *destino contém a rega solicitada (apenas se retorno == 1).
*/
int regas_obter_por_indice(int indice, Rega *destino);

/**
* @brief Devolve o número total de regas registadas.
*
* @return Quantidade de regas atualmente armazenadas.
* @post Valor de retorno >= 0.
*/
int regas_total(void);

#endif /* REGAS_H */

```

Ficheiro: include/tarefas.h

```
/**
```

```

* @file tarefas.h
* @brief Interface pública do módulo de Tarefas (GreenTrack).
*
* Este módulo gere tarefas associadas ao jardim comunitário,
* incluindo criação, conclusão, listagem por estado e
* persistência em ficheiro CSV.
*
* @author 2403882
* @version E-fólio B
* @date 2026
*/

#ifndef TAREFAS_H
#define TAREFAS_H

#include "tipos.h"

/**
 * @brief Inicializa o módulo de tarefas.
 *
 * Esvazia o array interno e repõe o contador de IDs automáticos.
 * Deve ser invocado uma única vez antes de qualquer outra operação.
 *
 * @post total_tarefas == 0 && proximo_id_tarefa == 1.
 */
void tarefas_inicializar(void);

/**
 * @brief Carrega tarefas a partir de ficheiro CSV.
 *
 * O formato esperado é: id_tarefa,descricao,data_prevista,concluida
 * por linha. A leitura interrompe-se se atingir MAX_TAREFAS.
 *
 * @param[in] nome_ficheiro Caminho para o ficheiro CSV.
 * @return 1 se a leitura foi bem-sucedida (mesmo que vazio),
 *         0 se o ficheiro não existe.
 * @pre nome_ficheiro != NULL.
 * @post O array interno contém as tarefas lidas; o próximo ID automático
 *       é recalculado como max(id_existentes)+1.
 */
int tarefas_carregar(const char *nome_ficheiro);

/**
 * @brief Guarda todas as tarefas em ficheiro CSV.
 *
 * @param[in] nome_ficheiro Caminho para o ficheiro de destino.
 * @return 1 se a escrita foi bem-sucedida,
 *         0 se ocorreu erro ao abrir o ficheiro.
 * @pre nome_ficheiro != NULL.
 * @post O ficheiro contém todas as tarefas em formato CSV.
 */

```

```

int tarefas_guardar(const char *nome_ficheiro);

/**
 * @brief Cria uma nova tarefa pendente.
 *
 * O ID é atribuído automaticamente e o estado inicial é sempre
 * não concluído (concluida = 0).
 *
 * @param[in] descricao Descrição textual da tarefa.
 * @param[in] data_prevista Dia previsto para conclusão.
 * @return 1 se a tarefa foi criada com sucesso,
 *         0 se o array está cheio.
 * @pre descricao != NULL.
 * @post A tarefa é adicionada ao array com concluida == 0;
 *        ficheiro tarefas.csv é reescrito.
 */
int tarefas_criar(const char *descricao, int data_prevista);

/**
 * @brief Marca uma tarefa como concluída.
 *
 * Procura a tarefa pelo ID e, se encontrada, altera o campo
 * concluida para 1, persistindo imediatamente.
 *
 * @param[in] id_tarefa Identificador da tarefa a concluir.
 * @return 1 se a tarefa foi encontrada e concluída,
 *         0 se nenhuma tarefa corresponde ao ID fornecido.
 * @pre id_tarefa > 0.
 * @post O campo concluida da tarefa passa a 1; ficheiro tarefas.csv
 *        é reescrito (apenas se a tarefa foi encontrada).
 */
int tarefas_concluir(int id_tarefa);

/**
 * @brief Lista no ecrã apenas as tarefas pendentes.
 *
 * Filtra as tarefas onde concluida == 0. Se não existirem,
 * apresenta mensagem informativa.
 *
 * @post Nenhuma alteração de estado.
 */
void tarefas_listar_pendentes(void);

/**
 * @brief Lista no ecrã todas as tarefas (concluídas e pendentes).
 *
 * Para cada tarefa, apresenta o ID, descrição, data prevista
 * e estado atual (Concluída / Pendente).
 *
 * @post Nenhuma alteração de estado.
 */

```

```

void tarefas_listar_todas(void);

/**
 * @brief Obtém uma cópia da tarefa num dado índice.
 *
 * @param[in]  indice    Índice no array interno (0-based).
 * @param[out] destino   Estrutura onde a cópia será escrita.
 * @return 1 se o índice é válido e a cópia foi efetuada,
 *         0 se o índice está fora dos limites.
 * @pre destino != NULL.
 * @post *destino contém a tarefa solicitada (apenas se retorno == 1).
 */
int tarefas_obter_por_indice(int indice, Tarefa *destino);

/**
 * @brief Devolve o número total de tarefas registadas.
 *
 * @return Quantidade de tarefas atualmente armazenadas.
 * @post Valor de retorno >= 0.
 */
int tarefas_total(void);

#endif /* TAREFAS_H */

```

## Ficheiro: include/io.h

```

/**
 * @file io.h
 * @brief Interface pública do módulo de Input/Output (GreenTrack).
 *
 * Este módulo centraliza todas as operações de interação com o
 * utilizador e com o sistema de ficheiros, garantindo leituras
 * robustas, limpeza de buffers e deteção de EOF.
 *
 * @author 2403882
 * @version E-fólio B
 * @date 2026
 */

#ifndef IO_H
#define IO_H

/**
 * @brief Lê uma string do teclado com limite de tamanho.
 *
 * Utiliza fgets() em vez de scanf("%s") para preservar espaços
 * internos na string. Remove automaticamente o caractere newline
 * e esvazia o buffer se o input exceder o tamanho máximo.
 *
 * @param[in]  prompt    Mensagem a apresentar ao utilizador.

```

```

* @param[out] destino      Buffer onde a string lida será escrita.
* @param[in]  tamanho_max  Capacidade do buffer (incluindo '\0').
* @pre destino != NULL && tamanho_max > 0.
* @post destino contém a string lida (truncada se necessário);
*       o buffer stdin fica vazio e pronto para a próxima leitura.
*/
void io_ler_string(const char *prompt, char *destino, int tamanho_max);

/**
* @brief Lê um inteiro do teclado.
*
* Apresenta o prompt, lê via scanf("%d") e esvazia o buffer
* para evitar que o newline residual interfira na leitura seguinte.
*
* @param[in] prompt  Mensagem a apresentar ao utilizador.
* @return O valor inteiro lido,
*        -1 se o input não for numérico,
*        -9999 se EOF for detetado.
* @post O buffer stdin fica vazio e pronto para a próxima leitura.
*/
int io_ler_inteiro(const char *prompt);

/**
* @brief Esvazia o buffer de entrada stdin.
*
* Descarta todos os caracteres até ao newline ou EOF.
* Substituto seguro de fflush(stdin), cujo comportamento é
* indefinido segundo o standard ISO C.
*
* @post Não existem caracteres residuais em stdin.
*/
void io_limpar_buffer(void);

/**
* @brief Apresenta o menu principal no ecrã.
*
* @param[in] data_atual  Dia atual do simulador.
* @post Nenhuma alteração de estado.
*/
void io_mostrar_menu(int data_atual);

/**
* @brief Pausa a execução até o utilizador pressionar Enter.
*
* @post O programa aguarda input do utilizador.
*/
void io_pausa(void);

/**
* @brief Verifica se foi detetado EOF em stdin.
*

```

```

* @return 1 se EOF foi atingido, 0 caso contrário.
* @post Nenhuma alteração de estado.
*/
int io_eof(void);

#endif /* IO_H */

```

## Ficheiro: src/main.c

```

/* =====
* main.c – Ponto de entrada e orquestração do GreenTrack
* GreenTrack – Gestão de Jardins Comunitários
*
* Este ficheiro implementa o padrão Controller: orquestra
* a interação entre os módulos de domínio (plantas, regas,
* tarefas) e o módulo de apresentação (io), sem conter
* lógica de negócio própria.
*
* Arquitetura do ciclo principal:
* 1. Inicialização: repõe contadores e carrega dados dos CSV.
* 2. Ciclo interativo: apresenta menu e delega ações aos
*    módulos respetivos.
* 3. Persistência de segurança: guarda todos os dados antes
*    de terminar, complementando a persistência imediata
*    já efetuada em cada operação de escrita.
*
* A data_atual utiliza um timestamp relativo (dia 0 =
* 01/01/2026), simplificando cálculos de intervalo.
* ===== */

#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 corresponde a 01/01/2026 */

    /* — Inicialização e carregamento de dados — */
    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 interativo ————— */
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:
        {
            /* Listar todas as plantas */
            plantas_listar();
            io_pausa();
            break;
        }

        case 2:
        {
            /* Adicionar nova planta ao sistema */
            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
            {
                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 para uma planta existente */
    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 quais plantas necessitam de rega */
    printf("\n--- Verificacao de Regas ---\n");
    plantas_verificar_rega(data_atual);
    io_pausa();
    break;
}

case 5:
{
    /* Listar tarefas pendentes */
    printf("\n");
    tarefas_listar_pendentes();
    io_pausa();
    break;
}

```



```

}

case 6:
{
    /* Criar nova tarefa pendente */
    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 pendente */
    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:
{
    /* Avançar simulação para o dia seguinte */
    data_atual++;
    printf("Data avancada para o dia %d.\n", data_atual);
}

```

```

        io_pausa();
        break;
    }

    case 0:
    {
        /* Terminar programa com persistência de segurança */
        printf("\nA guardar dados e a sair...\n");
        /* A persistência imediata em cada operação garante
           consistência, mas guardamos tudo explicitamente
           antes de sair para redundância. */
        plantas_guardar("plantas.csv");
        regas_guardar("regas.csv");
        tarefas_guardar("tarefas.csv");
        printf("Dados guardados com sucesso. Até logo!\n");
        break;
    }

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

} while (opcao != 0);

return 0;
}

```

Ficheiro: src/plantas.c

```

/**
 * @file plantas.c
 * @brief Implementação do módulo de gestão de plantas.
 *
 * Este módulo encapsula o conjunto de plantas do sistema GreenTrack.
 * O estado interno (array, contadores e gerador de IDs) é declarado
 * como static, garantindo que o acesso externo ocorra exclusivamente
 * através das funções públicas definidas em plantas.h.
 *
 * Decisões de arquitetura relevantes:
 * - Persistência imediata: cada operação de escrita sincroniza o
 *   estado com o ficheiro CSV, de modo que uma falha abrupta do
 *   programa não cause perda de dados.
 * - IDs automáticos com recuperação: após carregar de ficheiro,
 *   o próximo ID é recalculado como max(id_existente) + 1. Isto
 *   previne colisões se o CSV tiver sido editado externamente ou
 *   se registos foram eliminados, criando lacunas na sequência.
 */

```

```

* - Leitura via fscanf() em vez de feof(): o uso de feof() como
* condição de ciclo duplica o último registo porque o indicador
* EOF só é ativado após uma leitura falhar. A solução consiste
* em ler com fscanf() e interromper o ciclo quando o número de
* campos lidos for inferior ao esperado.
*/

#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) ————— */

/**
 * @brief Recalcula o maior ID existente e define o próximo ID disponível.
 *
 * Invocada após plantas_carregar() para garantir que novas plantas
 * recebam IDs únicos, mesmo que o ficheiro CSV contenha lacunas.
 */
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;
}

int plantas_carregar(const char *nome_ficheiro)
{
    FILE *f;
    int campos_lidos;

```

```

f = fopen(nome_ficheiro, "r");
if (f == NULL)
{
    /* Ficheiro inexistente: o sistema continua com array vazio.
    Não é tratado como erro fatal, pois pode ser a primeira
    execução ou o utilizador pode ter removido o ficheiro. */
    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);

/* Recalcular o próximo ID com base no maior existente, para
evitar colisões se o CSV tiver lacunas nos IDs. */
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;
}

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;
    }

    /* Guard clause: intervalo deve ser estritamente positivo */
    if (intervalo <= 0)
    {
        printf("Erro: intervalo de rega invalido (%d). Deve ser > 0.\n",
intervalo);
        return 0;
    }

    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;

    total_plantas++;
    proximo_id_planta++;

    /* Sincronização imediata com o ficheiro CSV, garantindo
       tolerância a falhas do processo. */
    plantas_guardar("plantas.csv");

    return 1;
}

```

```

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;

            /* Sincronização imediata: se o programa terminar
               inesperadamente, o estado da última rega não se perde. */
            plantas_guardar("plantas.csv");
            return 1;
        }
    }

    return 0;
}

int plantas_total(void)
{
    return total_plantas;
}

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;
}

```

## Ficheiro: 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.
 *
 * Decisões arquiteturais:
 * - Persistência imediata: cada rega é imediatamente escrita no
 *   ficheiro CSV, eliminando risco de perda de dados em crash.
 * - IDs automáticos: o próximo ID é sempre max(id_existentes)+1,
 *   garantindo unicidade mesmo após remoções.
 * - Leitura robusta: fscanf() em ciclo em vez de feof(), pois
 *   feof() só sinaliza EOF após uma tentativa de leitura falhada.
 * ===== */

#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) — */

/**
 * @brief Recalcula o próximo ID automático com base nos existentes.
 *
 * Itera sobre todas as regas carregadas para determinar o valor
 * máximo atual, evitando colisões de IDs após carregamento.
 */
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;
}

int regas_carregar(const char *nome_ficheiro)
{
    FILE *f;
    int campos_lidos;

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

    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;
}

int regas_registar(int id_planta, int data, int quantidade)
{
    Planta planta_tmp;

    /* Guard clause: verificar capacidade do array antes de avançar */
    if (total_regas >= MAX_REGAS)
    {
        printf("Erro: limite de regas atingido (%d).\n", MAX_REGAS);
        return 0;
    }

    /* Verificar integridade referencial: a planta deve existir */
    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;
}

```

```

    total_regas++;
    proximo_id_rega++;

    /* Atualizar ultima_rega da planta correspondente */
    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;
}

```

## Ficheiro: 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.
 *
 * Decisões arquiteturais:
 * - Persistência imediata: cada criação ou conclusão é imediatamente
 *   escrita no ficheiro CSV, eliminando risco de perda de dados.
 * - IDs automáticos: o próximo ID é sempre max(id_existentes)+1,
 *   garantindo unicidade mesmo após remoções.
 * - Estado binário: concluída ∈ {0, 1}, onde 0 = pendente e
 *   1 = concluída. Esta convenção simplifica filtros e listagens.
 * - Leitura robusta: fscanf() em ciclo em vez de feof(), pois
 *   feof() só sinaliza EOF após uma tentativa de leitura falhada.
 * ===== */

#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) — */

/**
 * @brief Recalcula o próximo ID automático com base nos existentes.
 *
 * Itera sobre todas as tarefas carregadas para determinar o valor
 * máximo atual, evitando colisões de IDs após carregamento.
 */
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;
}

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;
}

int tarefas_criar(const char *descricao, int data_prevista)
{
    /* Verificar capacidade do array antes de avançar */
    if (total_tarefas >= MAX_TAREFAS)
    {
        printf("Erro: limite de tarefas atingido (%d).\n", MAX_TAREFAS);
        return 0;
    }

    /* Validação de segurança: data prevista não pode ser negativa */
    if (data_prevista < 0)
    {
        printf("Erro: data prevista invalida (%d). Deve ser >= 0.\n",
data_prevista);
        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;
}

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;

            /* Persistir imediatamente para evitar perda de dados */
            tarefas_guardar("tarefas.csv");
            return 1;
        }
    }

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

void tarefas_listar_pendentes(void)
{
    int i;
    int encontrou = 0;

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

    for (i = 0; i < total_tarefas; i++)
    {
        /* Filtrar apenas tarefas com estado pendente (concluida == 0) */
        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 registada.\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;
}

```

**Ficheiro: src/io.c**

```

/* =====
 * io.c – Implementação do módulo de Input/Output
 * GreenTrack – Gestão de Jardins Comunitários
 *
 * Decisões arquiteturais de segurança de I/O:
 * - fgets() em vez de scanf("%s") para strings: preserva espaços
 *   internos e evita truncamento em nomes compostos.

```



```

* - Limpeza de buffer via getchar() em vez de fflush(stdin):
*   fflush(stdin) tem comportamento indefinido em ISO C;
*   o ciclo while(getchar()) é portátil e previsível.
* - Detecção de EOF: evita loops infinitos quando o programa
*   é executado em modo pipe ou com redirecionamento de ficheiro.
* ===== */

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

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);

    /* Remover newline se presente */
    if (len > 0 && destino[len - 1] == '\n')
    {
        destino[len - 1] = '\0';
    }
    else if (len > 0)
    {
        /* Input excedeu o buffer – descartar caracteres residuais */
        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 */
    }

    /* Limpar buffer após leitura numérica: o newline residual
       de scanf("%d") iria ser consumido pelo próximo fgets(),
       resultando em string vazia. */
    io_limpar_buffer();

    if (resultado != 1)
    {
        return -1;
    }

    return valor;
}

void io_limpar_buffer(void)
{
    int c;

    /* Descartar todos os caracteres até newline ou EOF.
       fflush(stdin) é indefinido em ISO C; este ciclo é
       portátil e garante um estado limpo do buffer. */
    while ((c = getchar()) != '\n' && c != EOF)
    {
        /* descartar caractere */
    }
}

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 caracteres até newline */
    }
}

```

#### Ficheiro: testes/testes\_unidade.c

```

/* =====
 * testes_unidade.c – Testes de unidade para o GreenTrack
 * RF 4.3: Testes sistemáticos – padrão printf() + assert()
 *
 * Cenários testados por função:
 * [Normal] – valores típicos de utilização
 * [Limite] – fronteiras do domínio (N±1)
 * [Erro] – violações de pré-condição ou condições inválidas
 * ===== */

#include <stdio.h>
#include <assert.h>
#include <string.h>
#include "tipos.h"
#include "plantas.h"
#include "regas.h"
#include "tarefas.h"

/* =====
 * TESTES: plantas_adicionar()
 * ===== */

/**
 * [Normal] Adicionar uma planta com parâmetros válidos.
 * Verifica: retorno == 1; total incrementado; ID atribuído.
 */
void teste_plantas_adicionar_normal(void)
{
    int obtido, esperado;
    Planta p;

    printf("TESTE: plantas_adicionar() [Normal] – adicionar planta valida\n");

    plantas_inicializar();

```

```

    obtido = plantas_adicionar("Rosa", "Rosa chinensis", "15/03/2026", 7, 0);
    esperado = 1;
    if (obtido != esperado)
        printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
    assert(obtido == esperado);

    /* Verificar pós-condição: total == 1 */
    if (plantas_total() != 1)
        printf(" FALHA: total esperado 1, obtido %d\n", plantas_total());
    assert(plantas_total() == 1);

    /* Verificar pós-condição: ID atribuído automaticamente */
    if (plantas_obter_por_indice(0, &p) != 1)
        printf(" FALHA: nao conseguiu obter planta no indice 0\n");
    assert(plantas_obter_por_indice(0, &p) == 1);

    if (p.id != 1)
        printf(" FALHA: id esperado 1, obtido %d\n", p.id);
    assert(p.id == 1);

    /* Verificar pós-condição: ultima_rega == data_atual (0) */
    if (p.ultima_rega != 0)
        printf(" FALHA: ultima_rega esperado 0, obtido %d\n", p.ultima_rega);
    assert(p.ultima_rega == 0);

    printf(" OK\n");
}

/**
 * [Limite] Adicionar até ao limite MAX_PLANTAS (100).
 * Verifica: a última adição retorna 1; a seguinte retorna 0.
 */
void teste_plantas_adicionar_limite(void)
{
    int obtido, esperado;
    int i;

    printf("TESTE: plantas_adicionar() [Limite] – encher array ate MAX_PLANTAS (%d)\n", MAX_PLANTAS);

    plantas_inicializar();

    /* Preencher até MAX_PLANTAS - 1 */
    for (i = 0; i < MAX_PLANTAS - 1; i++)
    {
        obtido = plantas_adicionar("Planta", "Especie", "01/01/2026", 7, 0);
        if (obtido != 1)
        {
            printf(" FALHA: falhou ao adicionar planta %d (esperado 1, obtido %d)\n", i + 1, obtido);

```

```

        assert(obtido == 1);
    }
}

/* Adicionar a última (MAX_PLANTAS) – deve ter sucesso */
obtido = plantas_adicionar("Ultima", "Especie", "01/01/2026", 7, 0);
esperado = 1;
if (obtido != esperado)
    printf(" FALHA: adicionar planta %d esperado %d, obtido %d\n",
MAX_PLANTAS, esperado, obtido);
assert(obtido == esperado);

if (plantas_total() != MAX_PLANTAS)
    printf(" FALHA: total esperado %d, obtido %d\n", MAX_PLANTAS,
plantas_total());
assert(plantas_total() == MAX_PLANTAS);

/* Tentar adicionar além do limite – deve falhar */
obtido = plantas_adicionar("Extra", "Especie", "01/01/2026", 7, 0);
esperado = 0;
if (obtido != esperado)
    printf(" FALHA: adicionar alem do limite esperado %d, obtido %d\n",
esperado, obtido);
assert(obtido == esperado);

/* Verificar que o total não aumentou (preservação de estado) */
if (plantas_total() != MAX_PLANTAS)
    printf(" FALHA: total apos falha esperado %d, obtido %d\n", MAX_PLANTAS,
plantas_total());
assert(plantas_total() == MAX_PLANTAS);

printf(" OK\n");
}

/**
 * [Erro] Intervalo de rega inválido (<= 0).
 * Verifica: retorno == 0; estado preservado.
 */
void teste_plantas_adicionar_erro_intervalo(void)
{
    int obtido, esperado;
    int total_antes;
    Planta p;

    printf("TESTE: plantas_adicionar() [Erro] – intervalo invalido (<= 0)\n");

    plantas_inicializar();
    plantas_adicionar("Rosa", "Rosa chinensis", "15/03/2026", 7, 0);
    total_antes = plantas_total();

    obtido = plantas_adicionar("Orquidea", "Orchidaceae", "20/03/2026", 0, 0);

```

```

    esperado = 0;
    if (obtido != esperado)
        printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
    assert(obtido == esperado);

    /* Preservação: total não deve ter aumentado */
    if (plantas_total() != total_antes)
        printf(" FALHA: total preservado esperado %d, obtido %d\n", total_antes,
plantas_total());
    assert(plantas_total() == total_antes);

    /* Verificar que a planta válida anterior ainda existe */
    if (plantas_obter_por_indice(0, &p) != 1)
        printf(" FALHA: planta anterior nao encontrada\n");
    assert(plantas_obter_por_indice(0, &p) == 1);
    assert(p.id == 1);

    printf(" OK\n");
}

/* =====
 * TESTES: plantas_obter_por_id()
 * ===== */

/**
 * [Normal] Procurar planta existente.
 * Verifica: retorno >= 0; dados copiados corretamente.
 */
void teste_plantas_obter_por_id_normal(void)
{
    int indice;
    Planta p;

    printf("TESTE: plantas_obter_por_id() [Normal] – procurar planta existente\n");

    plantas_inicializar();
    plantas_adicionar("Rosa", "Rosa chinensis", "15/03/2026", 7, 0);

    indice = plantas_obter_por_id(1, &p);
    if (indice < 0)
        printf(" FALHA: indice esperado >= 0, obtido %d\n", indice);
    assert(indice >= 0);

    if (strcmp(p.nome, "Rosa") != 0)
        printf(" FALHA: nome esperado 'Rosa', obtido '%s'\n", p.nome);
    assert(strcmp(p.nome, "Rosa") == 0);

    printf(" OK\n");
}

/**

```

```

* [Erro] Procurar planta inexistente.
* Verificar: retorno == -1.
*/
void teste_plantas_obter_por_id_erro(void)
{
    int indice;
    Planta p;

    printf("TESTE: plantas_obter_por_id() [Erro] – procurar planta inexistente\n");

    plantas_inicializar();

    indice = plantas_obter_por_id(999, &p);
    if (indice != -1)
        printf(" FALHA: retorno esperado -1, obtido %d\n", indice);
    assert(indice == -1);

    printf(" OK\n");
}

/**
* [Limite] Procurar planta no ID máximo após encher o array.
* Verifica: retorno >= 0; dados corretos do último ID.
*/
void teste_plantas_obter_por_id_limite(void)
{
    int indice;
    Planta p;
    int i;

    printf("TESTE: plantas_obter_por_id() [Limite] – procurar ID maximo apos encher
array\n");

    plantas_inicializar();

    /* Preencher até MAX_PLANTAS */
    for (i = 0; i < MAX_PLANTAS; i++)
    {
        plantas_adicionar("Planta", "Especie", "01/01/2026", 7, 0);
    }

    /* O último ID gerado é MAX_PLANTAS */
    indice = plantas_obter_por_id(MAX_PLANTAS, &p);
    if (indice < 0)
        printf(" FALHA: indice esperado >= 0, obtido %d\n", indice);
    assert(indice >= 0);

    if (p.id != MAX_PLANTAS)
        printf(" FALHA: id esperado %d, obtido %d\n", MAX_PLANTAS, p.id);
    assert(p.id == MAX_PLANTAS);
}

```

```

    printf(" OK\n");
}

/* =====
 * TESTES: regas_registar()
 * ===== */

/**
 * [Normal] Registrar rega para planta existente.
 * Verifica: retorno == 1; total_regas incrementado.
 */
void teste_regas_registar_normal(void)
{
    int obtido, esperado;
    Rega r;
    Planta p;

    printf("TESTE: regas_registar() [Normal] – regar planta existente\n");

    plantas_inicializar();
    regas_inicializar();
    plantas_adicionar("Rosa", "Rosa chinensis", "15/03/2026", 7, 0);

    obtido = regas_registar(1, 5, 500);
    esperado = 1;
    if (obtido != esperado)
        printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
    assert(obtido == esperado);

    if (regas_total() != 1)
        printf(" FALHA: total_regas esperado 1, obtido %d\n", regas_total());
    assert(regas_total() == 1);

    /* Verificar integridade referencial: id_planta == 1 */
    if (regas_obter_por_indice(0, &r) != 1)
        printf(" FALHA: nao conseguiu obter rega no indice 0\n");
    assert(regas_obter_por_indice(0, &r) == 1);
    assert(r.id_planta == 1);
    assert(r.quantidade_agua == 500);

    /* Verificar que ultima_rega da planta foi atualizada */
    plantas_obter_por_id(1, &p);
    if (p.ultima_rega != 5)
        printf(" FALHA: ultima_rega esperado 5, obtido %d\n", p.ultima_rega);
    assert(p.ultima_rega == 5);

    printf(" OK\n");
}

/**
 * [Erro] Registrar rega para planta inexistente.

```



```

/* Verifica: retorno == 0; estado preservado.
*/
void teste_regas_registar_erro_planta(void)
{
    int obtido, esperado;
    int total_antes;

    printf("TESTE: regas_registar() [Erro] - planta inexistente\n");

    plantas_inicializar();
    regas_inicializar();
    plantas_adicionar("Rosa", "Rosa chinensis", "15/03/2026", 7, 0);

    total_antes = regas_total();

    obtido = regas_registar(999, 5, 500);
    esperado = 0;
    if (obtido != esperado)
        printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
    assert(obtido == esperado);

    /* Preservação: total_regas não deve ter aumentado */
    if (regas_total() != total_antes)
        printf(" FALHA: total_regas preservado esperado %d, obtido %d\n",
total_antes, regas_total());
    assert(regas_total() == total_antes);

    printf(" OK\n");
}

/**
 * [Limite] Encher array de regas até MAX_REGAS (500).
 * Verifica: a 501.ª rega é rejeitada defensivamente.
 */
void teste_regas_registar_limite(void)
{
    int obtido, esperado;
    int i;

    printf("TESTE: regas_registar() [Limite] - encher array ate MAX_REGAS (%d)\n",
MAX_REGAS);

    plantas_inicializar();
    regas_inicializar();
    plantas_adicionar("Rosa", "Rosa chinensis", "15/03/2026", 7, 0);

    /* Preencher até MAX_REGAS - 1 */
    for (i = 0; i < MAX_REGAS - 1; i++)
    {
        obtido = regas_registar(1, i % 30, 500);
        if (obtido != 1)

```

```

        {
            printf(" FALHA: falhou ao registrar rega %d (esperado 1, obtido %d)\n",
i + 1, obtido);
            assert(obtido == 1);
        }
    }

    /* Última rega válida */
    obtido = regas_registar(1, 30, 500);
    esperado = 1;
    if (obtido != esperado)
        printf(" FALHA: registrar rega %d esperado %d, obtido %d\n", MAX_REGAS,
esperado, obtido);
    assert(obtido == esperado);

    if (regas_total() != MAX_REGAS)
        printf(" FALHA: total esperado %d, obtido %d\n", MAX_REGAS,
regas_total());
    assert(regas_total() == MAX_REGAS);

    /* Tentar registrar além do limite */
    obtido = regas_registar(1, 31, 500);
    esperado = 0;
    if (obtido != esperado)
        printf(" FALHA: registrar alem do limite esperado %d, obtido %d\n",
esperado, obtido);
    assert(obtido == esperado);

    /* Verificar que o total não aumentou */
    if (regas_total() != MAX_REGAS)
        printf(" FALHA: total apos falha esperado %d, obtido %d\n", MAX_REGAS,
regas_total());
    assert(regas_total() == MAX_REGAS);

    printf(" OK\n");
}

/* =====
 * TESTES: tarefas_criar()
 * ===== */

/**
 * [Normal] Criar tarefa com descrição válida.
 * Verifica: retorno == 1; estado inicial concluida == 0.
 */
void teste_tarefas_criar_normal(void)
{
    int obtido, esperado;
    Tarefa t;

    printf("TESTE: tarefas_criar() [Normal] - criar tarefa valida\n");

```

```

tarefas_inicializar();

obtido = tarefas_criar("Regar rosas", 10);
esperado = 1;
if (obtido != esperado)
    printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
assert(obtido == esperado);

if (tarefas_total() != 1)
    printf(" FALHA: total esperado 1, obtido %d\n", tarefas_total());
assert(tarefas_total() == 1);

/* Verificar pós-condição: concluida == 0 */
if (tarefas_obter_por_indice(0, &t) != 1)
    printf(" FALHA: nao conseguiu obter tarefa no indice 0\n");
assert(tarefas_obter_por_indice(0, &t) == 1);

if (t.concluida != 0)
    printf(" FALHA: concluida esperado 0, obtido %d\n", t.concluida);
assert(t.concluida == 0);

if (strcmp(t.descricao, "Regar rosas") != 0)
    printf(" FALHA: descricao esperada 'Regar rosas', obtida '%s'\n",
t.descricao);
assert(strcmp(t.descricao, "Regar rosas") == 0);

printf(" OK\n");
}

/**
 * [Limite] Criar até ao limite MAX_TAREFAS (200).
 * Verifica: última criação retorna 1; a seguinte retorna 0.
 */
void teste_tarefas_criar_limite(void)
{
    int obtido, esperado;
    int i;

    printf("TESTE: tarefas_criar() [Limite] – encher array ate MAX_TAREFAS (%d)\n",
MAX_TAREFAS);

    tarefas_inicializar();

    for (i = 0; i < MAX_TAREFAS - 1; i++)
    {
        obtido = tarefas_criar("Tarefa", i);
        if (obtido != 1)
        {
            printf(" FALHA: falhou ao criar tarefa %d (esperado 1, obtido %d)\n",
i + 1, obtido);

```

```

        assert(obtido == 1);
    }
}

/* Última tarefa válida */
obtido = tarefas_criar("Ultima", 999);
esperado = 1;
if (obtido != esperado)
    printf(" FALHA: criar tarefa %d esperado %d, obtido %d\n", MAX_TAREFAS,
esperado, obtido);
    assert(obtido == esperado);

    if (tarefas_total() != MAX_TAREFAS)
        printf(" FALHA: total esperado %d, obtido %d\n", MAX_TAREFAS,
tarefas_total());
    assert(tarefas_total() == MAX_TAREFAS);

/* Tentar criar além do limite */
obtido = tarefas_criar("Extra", 1000);
esperado = 0;
if (obtido != esperado)
    printf(" FALHA: criar alem do limite esperado %d, obtido %d\n", esperado,
obtido);
    assert(obtido == esperado);

    if (tarefas_total() != MAX_TAREFAS)
        printf(" FALHA: total apos falha esperado %d, obtido %d\n", MAX_TAREFAS,
tarefas_total());
    assert(tarefas_total() == MAX_TAREFAS);

    printf(" OK\n");
}

/**
 * [Erro] Passar data prevista inválida (negativa).
 * Verifica: retorno == 0; estado preservado.
 */
void teste_tarefas_criar_erro_data(void)
{
    int obtido, esperado;
    int total_antes;
    Tarefa t;

    printf("TESTE: tarefas_criar() [Erro] – data prevista invalida (< 0)\n");

    tarefas_inicializar();
    tarefas_criar("Regar rosas", 10);
    total_antes = tarefas_total();

    obtido = tarefas_criar("Tarefa invalida", -1);
    esperado = 0;

```

```

    if (obtido != esperado)
        printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
    assert(obtido == esperado);

    /* Preservação: total não deve ter aumentado */
    if (tarefas_total() != total_antes)
        printf(" FALHA: total preservado esperado %d, obtido %d\n", total_antes,
tarefas_total());
    assert(tarefas_total() == total_antes);

    /* Verificar que a tarefa anterior continua íntegra */
    if (tarefas_obter_por_indice(0, &t) != 1)
        printf(" FALHA: tarefa anterior nao encontrada\n");
    assert(tarefas_obter_por_indice(0, &t) == 1);
    assert(t.data_prevista == 10);

    printf(" OK\n");
}

/* =====
 * TESTES: tarefas_concluir()
 * ===== */

/**
 * [Normal] Concluir tarefa existente.
 * Verifica: retorno == 1; concluida passa a 1.
 */
void teste_tarefas_concluir_normal(void)
{
    int obtido, esperado;
    Tarefa t;

    printf("TESTE: tarefas_concluir() [Normal] – concluir tarefa existente\n");

    tarefas_inicializar();
    tarefas_criar("Regar rosas", 10);

    obtido = tarefas_concluir(1);
    esperado = 1;
    if (obtido != esperado)
        printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
    assert(obtido == esperado);

    /* Verificar pós-condição: concluida == 1 */
    if (tarefas_obter_por_indice(0, &t) != 1)
        printf(" FALHA: nao conseguiu obter tarefa no indice 0\n");
    assert(tarefas_obter_por_indice(0, &t) == 1);

    if (t.concluida != 1)
        printf(" FALHA: concluida esperado 1, obtido %d\n", t.concluida);
    assert(t.concluida == 1);

```

```

    printf(" OK\n");
}

/**
 * [Erro] Concluir tarefa inexistente.
 * Verifica: retorno == 0; estado preservado.
 */
void teste_tarefas_concluir_erro(void)
{
    int obtido, esperado;
    int total_antes;
    Tarefa t;

    printf("TESTE: tarefas_concluir() [Erro] - tarefa inexistente\n");

    tarefas_inicializar();
    tarefas_criar("Regar rosas", 10);
    total_antes = tarefas_total();

    obtido = tarefas_concluir(999);
    esperado = 0;
    if (obtido != esperado)
        printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
    assert(obtido == esperado);

    /* Preservação: total não alterado; tarefa 1 continua pendente */
    if (tarefas_total() != total_antes)
        printf(" FALHA: total preservado esperado %d, obtido %d\n", total_antes,
tarefas_total());
    assert(tarefas_total() == total_antes);

    if (tarefas_obter_por_indice(0, &t) != 1)
        printf(" FALHA: tarefa original nao encontrada\n");
    assert(tarefas_obter_por_indice(0, &t) == 1);
    assert(t.concluida == 0); /* continua pendente */

    printf(" OK\n");
}

/**
 * [Limite] Concluir a tarefa na última posição possível do array.
 * Verifica: retorno == 1; concluida == 1.
 */
void teste_tarefas_concluir_limite(void)
{
    int obtido, esperado;
    Tarefa t;
    int i;

```

```

    printf("TESTE: tarefas_concluir() [Limite] – concluir tarefa na ultima posicao
(%d)\n", MAX_TAREFAS);

    tarefas_inicializar();

    /* Preencher até MAX_TAREFAS */
    for (i = 0; i < MAX_TAREFAS; i++)
    {
        tarefas_criar("Tarefa", i);
    }

    /* Concluir a última tarefa (ID = MAX_TAREFAS) */
    obtido = tarefas_concluir(MAX_TAREFAS);
    esperado = 1;
    if (obtido != esperado)
        printf(" FALHA: retorno esperado %d, obtido %d\n", esperado, obtido);
    assert(obtido == esperado);

    /* Verificar pós-condição: última tarefa concluída */
    if (tarefas_obter_por_indice(MAX_TAREFAS - 1, &t) != 1)
        printf(" FALHA: nao conseguiu obter tarefa no indice %d\n", MAX_TAREFAS -
1);
    assert(tarefas_obter_por_indice(MAX_TAREFAS - 1, &t) == 1);

    if (t.concluida != 1)
        printf(" FALHA: concluida esperado 1, obtido %d\n", t.concluida);
    assert(t.concluida == 1);

    if (t.id_tarefa != MAX_TAREFAS)
        printf(" FALHA: id_tarefa esperado %d, obtido %d\n", MAX_TAREFAS,
t.id_tarefa);
    assert(t.id_tarefa == MAX_TAREFAS);

    printf(" OK\n");
}

/* =====
 * MAIN – Orquestração dos testes
 * ===== */

int main(void)
{
    printf("=====\n");
    printf(" Testes de Unidade – GreenTrack\n");
    printf(" Padrão: printf() + assert()\n");
    printf("=====\n\n");

    /* --- plantas_adicionar() --- */
    teste_plantas_adicionar_normal();
    teste_plantas_adicionar_limite();
    teste_plantas_adicionar_erro_intervalo();

```

```

/* --- plantas_obter_por_id() --- */
teste_plantas_obter_por_id_normal();
teste_plantas_obter_por_id_erro();
teste_plantas_obter_por_id_limite();

/* --- regas_registar() --- */
teste_regas_registar_normal();
teste_regas_registar_erro_planta();
teste_regas_registar_limite();

/* --- tarefas_criar() --- */
teste_tarefas_criar_normal();
teste_tarefas_criar_limite();
teste_tarefas_criar_erro_data();

/* --- tarefas_concluir() --- */
teste_tarefas_concluir_normal();
teste_tarefas_concluir_erro();
teste_tarefas_concluir_limite();

printf("\n=====\\n");
printf("  Todos os testes passaram com sucesso!\\n");
printf("=====\\n");

return 0;
}

```

#### Ficheiro: testes/testes\_integracao.c

```

/* =====
 * testes_integracao.c – Testes de integração para o GreenTrack
 * RF 4.3: Testes sistemáticos – padrão printf() + assert()
 *
 * Cenários de integração:
 *   [A] regas_registar() → atualização de ultima_rega em plantas
 *       (cooperação entre módulos regas e plantas)
 *   [B] tarefas_concluir() → persistência imediata em CSV
 *       (ciclo: criar → concluir → guardar → inicializar → carregar → verificar)
 *   [C] plantas_carregar() com CSV inexistente/corrompido
 *       (robustez do módulo face a dados externos malformados)
 * ===== */

#include <stdio.h>
#include <assert.h>
#include <string.h>
#include "tipos.h"
#include "plantas.h"
#include "regas.h"
#include "tarefas.h"

```



```

/* =====
 * CENÁRIO A: Cooperação regas → plantas
 * Verificar se regas_registar() atualiza ultima_rega da planta
 * ===== */

/**
 * [Integração A] Registrar rega e verificar atualização cruzada.
 *
 * Fluxo: adicionar planta → registrar rega → verificar:
 * 1. total_regas incrementado
 * 2. ultima_rega da planta atualizado para data da rega
 * 3. integridade referencial (id_planta correto na rega)
 */
void teste_integracao_rega_atualiza_planta(void)
{
    int obtido;
    Planta p;
    Rega r;

    printf("TESTE [Integracao A]: regas_registar() → atualiza ultima_rega em plantas\n");

    /* Arrange: estado limpo + 1 planta */
    plantas_inicializar();
    regas_inicializar();
    plantas_adicionar("Rosa", "Rosa chinensis", "15/03/2026", 7, 0);

    /* Act: registrar rega no dia 5 */
    obtido = regas_registar(1, 5, 500);

    /* Assert: rega registada com sucesso */
    if (obtido != 1)
        printf(" FALHA: regas_registar() retornou %d, esperado 1\n", obtido);
    assert(obtido == 1);

    /* Assert: total de regas incrementado */
    if (regas_total() != 1)
        printf(" FALHA: total_regas = %d, esperado 1\n", regas_total());
    assert(regas_total() == 1);

    /* Assert: ultima_rega da planta foi atualizada para dia 5 */
    plantas_obter_por_id(1, &p);
    if (p.ultima_rega != 5)
        printf(" FALHA: ultima_rega = %d, esperado 5\n", p.ultima_rega);
    assert(p.ultima_rega == 5);

    /* Assert: integridade referencial da rega */
    regas_obter_por_indice(0, &r);
    if (r.id_planta != 1)
        printf(" FALHA: id_planta na rega = %d, esperado 1\n", r.id_planta);
}

```

```

    assert(r.id_planta == 1);

    if (r.quantidade_agua != 500)
        printf(" FALHA: quantidade = %d, esperado 500\n", r.quantidade_agua);
    assert(r.quantidade_agua == 500);

    printf(" OK\n");
}

/* =====
 * CENÁRIO B: Persistência imediata de tarefas
 * Verificar se tarefas_concluir() persiste corretamente em CSV
 * ===== */

/**
 * [Integração B] Ciclo completo de persistência de tarefas.
 *
 * Fluxo: criar tarefa → concluir → guardar (automático) →
 *         inicializar memória → carregar de CSV → verificar estado.
 */
void teste_integracao_persistencia_tarefas(void)
{
    int obtido;
    Tarefa t;

    printf("TESTE [Integracao B]: tarefas_concluir() → persistencia em
tarefas.csv\n");

    /* Arrange: estado limpo + 1 tarefa pendente */
    tarefas_inicializar();
    tarefas_criar("Regar rosas", 10);

    /* Verificar estado inicial */
    tarefas_obter_por_indice(0, &t);
    if (t.concluida != 0)
        printf(" FALHA: concluida inicial = %d, esperado 0\n", t.concluida);
    assert(t.concluida == 0);

    /* Act: concluir tarefa (isto dispara tarefas_guardar automaticamente) */
    obtido = tarefas_concluir(1);
    if (obtido != 1)
        printf(" FALHA: tarefas_concluir() retornou %d, esperado 1\n", obtido);
    assert(obtido == 1);

    /* Verificar estado em memória após conclusão */
    tarefas_obter_por_indice(0, &t);
    if (t.concluida != 1)
        printf(" FALHA: concluida apos concluir = %d, esperado 1\n", t.concluida);
    assert(t.concluida == 1);

    /* Act: simular reinício do programa – limpar memória e recarregar */

```

```

tarefas_inicializar();
tarefas_carregar("tarefas.csv");

/* Assert: estado recuperado do CSV */
if (tarefas_total() != 1)
    printf(" FALHA: total apos carregar = %d, esperado 1\n", tarefas_total());
assert(tarefas_total() == 1);

tarefas_obter_por_indice(0, &t);
if (t.concluida != 1)
    printf(" FALHA: concluida apos carregar = %d, esperado 1\n", t.concluida);
assert(t.concluida == 1);

if (strcmp(t.descricao, "Regar rosas") != 0)
    printf(" FALHA: descricao = '%s', esperado 'Regar rosas'\n", t.descricao);
assert(strcmp(t.descricao, "Regar rosas") == 0);

if (t.data_prevista != 10)
    printf(" FALHA: data_prevista = %d, esperado 10\n", t.data_prevista);
assert(t.data_prevista == 10);

printf(" OK\n");
}

/* =====
 * CENÁRIO C: Robustez face a CSV inexistente/corrompido
 * ===== */

/**
 * [Integração C] Carregar ficheiro CSV inexistente.
 *
 * Verifica se o módulo plantas lida de forma defensiva com a
 * ausência do ficheiro, retornando 0 sem crashar.
 */
void teste_integracao_csv_inexistente(void)
{
    int obtido;

    printf("TESTE [Integracao C1]: plantas_carregar() com CSV inexistente\n");

    /* Garantir que o ficheiro não existe */
    remove("plantas_inexistente.csv");

    /* Act */
    plantas_inicializar();
    obtido = plantas_carregar("plantas_inexistente.csv");

    /* Assert: retorno 0 (não é erro fatal), estado vazio preservado */
    if (obtido != 0)
        printf(" FALHA: retorno = %d, esperado 0\n", obtido);
    assert(obtido == 0);
}

```

```

    if (plantas_total() != 0)
        printf(" FALHA: total = %d, esperado 0\n", plantas_total());
    assert(plantas_total() == 0);

    printf(" OK\n");
}

/**
 * [Integração C2] Carregar ficheiro CSV corrompido.
 *
 * Cria um CSV com uma linha válida e uma linha malformada.
 * Verifica se o módulo para na linha corrompida sem crashar
 * e preserva os registos lidos anteriormente.
 */
void teste_integracao_csv_corrompido(void)
{
    FILE *f;
    int obtido;
    Planta p;

    printf("TESTE [Integracao C2]: plantas_carregar() com CSV corrompido\n");

    /* Arrange: criar CSV com linha válida + linha malformada */
    f = fopen("plantas_corrompido.csv", "w");
    if (f == NULL)
    {
        printf(" AVISO: nao foi possivel criar ficheiro de teste\n");
        return;
    }
    fprintf(f, "1,Rosa,Rosa chinensis,15/03/2026,7,0\n");
    fprintf(f, "2,Margarida,Malformada\n"); /* linha corrompida: só 3 campos */
    fclose(f);

    /* Act */
    plantas_inicializar();
    obtido = plantas_carregar("plantas_corrompido.csv");

    /* Assert: carregou com sucesso (ficheiro existe) */
    if (obtido != 1)
        printf(" FALHA: retorno = %d, esperado 1\n", obtido);
    assert(obtido == 1);

    /* Assert: leu apenas a linha válida, parou na malformada */
    if (plantas_total() != 1)
        printf(" FALHA: total = %d, esperado 1\n", plantas_total());
    assert(plantas_total() == 1);

    /* Assert: integridade do registo válido preservada */
    plantas_obter_por_indice(0, &p);
    if (p.id != 1)

```

```

        printf(" FALHA: id = %d, esperado 1\n", p.id);
        assert(p.id == 1);

        if (strcmp(p.nome, "Rosa") != 0)
            printf(" FALHA: nome = '%s', esperado 'Rosa'\n", p.nome);
        assert(strcmp(p.nome, "Rosa") == 0);

        /* Limpeza */
        remove("plantas_corrompido.csv");

        printf(" OK\n");
    }

    /* =====
    * MAIN – Orquestração dos testes de integração
    * ===== */

int main(void)
{
    printf("=====\n");
    printf(" Testes de Integracao – GreenTrack\n");
    printf(" Padrão: printf() + assert()\n");
    printf("=====\n\n");

    teste_integracao_rega_atualiza_planta();
    teste_integracao_persistencia_tarefas();
    teste_integracao_csv_inexistente();
    teste_integracao_csv_corrompido();

    printf("\n=====\n");
    printf(" Todos os testes de integracao passaram!\n");
    printf("=====\n");

    return 0;
}

```