E-fólio A | Instruções para a realização do E-fólio

# LABORATÓRIO DE PROGRAMAÇÃO | 21178

## Período de realização

Decorre de 10 de abril a 19 de abril de 2026

## Data limite de entrega

19 de abril de 2026, até às 23:59 de Portugal Continental

## Temática

Deteção e correção de erros & Organização modular de código

## Objetivos

- Identificar e corrigir erros em código existente (Módulo 1);
- Reorganizar código monolítico numa arquitetura modular (Módulo 2);
- Documentar decisões e justificar alterações.

## Cenário: GreenTrack - Gestão de Jardins Comunitários

A GreenTrack é uma plataforma para gestão de jardins comunitários. O sistema permite registar plantas, controlar regas e gerir tarefas de manutenção. O programa garante que os dados são persistidos em ficheiros csv e recuperados numa execução posterior.

Um programador iniciante desenvolveu um programa monolítico em C. O código contém vários erros e está desorganizado. Para este trabalho, em vez de receber o ficheiro do projeto completo, são-lhe fornecidos excertos com contexto mínimo, com linhas suficientes para compreender o problema, mas não mais do que isso.

O objetivo deste trabalho é duplo:

1. **Corrigir os erros** presentes nos excertos, documentando cada um deles. Existem vários erros (mais de 10), não sendo indicado o número exato.
2. **Escrever o código completo corrigido e modularizado:** o código final é inteiramente da sua autoria, devendo produzir uma versão final coerente do sistema, devendo reutilizar e adaptar os excertos fornecidos. O sistema deve permitir:

   a) listar plantas;
   b) registar regas;
   c) criar e concluir tarefas;
   d) guardar/carregar dados.

O código final deve garantir que, após qualquer operação de escrita (adicionar planta, registar rega, criar tarefa), os dados persistidos em disco refletem imediatamente a alteração, exceto quando indicado em contrário, por opção do utilizador.

O trabalho deve incidir sobre a correção e melhoria do código fornecido. Não é esperado nem valorizado reescrever integralmente a solução, mas sim identificar, justificar e corrigir os problemas existentes.

**Nota importante**: O código deve ser entregue em dois formatos: incluído em texto editável no relatório (PDF) e em ficheiro executável (ZIP).

## Contexto: Tipos de dados e variáveis globais

Os tipos e variáveis abaixo são partilhados por todo o programa. Servem de referência para interpretar os excertos seguintes.

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

/* ── Tipos de dados ──────────────────────────────────────── */

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

typedef struct {
    int  id;
    char nome[50];
    char especie[50];
    char data_plantio[11];   /* DD/MM/AAAA */
    int  intervalo_rega;     /* em dias */
    int  ultima_rega;        /* timestamp: dias desde 01/01/2026 */
} Planta;

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

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

Planta  plantas[MAX_PLANTAS];
Rega    regas[MAX_REGAS];
Tarefa  tarefas[MAX_TAREFAS];

int total_plantas = 0;
int total_regas   = 0;
int total tarefas = 0;
```

## Excertos com erros

**Modelo de datas:** ao longo do código, as datas são representadas como *int* com um *timestamp simplificado:* o número de dias decorridos desde 01/01/2026. Por exemplo, 05/01/2026 corresponde ao valor 4 e 01/02/2026 corresponde ao valor 31. Esta convenção aplica-se aos campos *ultima_rega*, *data_rega* e *data_prevista*, bem como ao parâmetro *data_atual* passado às funções de verificação.

Cada excerto mostra apenas o contexto mínimo necessário para identificar o problema. As secções omitidas estão marcadas com *[...]*. O número total de erros não é indicado, encontrá-los todos faz parte da avaliação.

As perguntas de introdução de cada excerto não necessitam de ser respondidas em relatório.

**Excerto 1: carregar_dados()**

*Observe o ciclo de leitura de plantas.csv e a condição que controla quando parar de ler.*

```
void carregar_dados() {
    FILE *f_plantas = fopen("plantas.csv", "r");
    if (f_plantas == NULL) { printf("Erro\n"); return; }

    while (!feof(f_plantas)) {
        fscanf(f_plantas, "%d,%[^,],%[^,],%[^,],%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);
        total_plantas++;
    }
    fclose(f_plantas);
    FILE *f_regas = fopen("regas.csv", "r");
    while (!feof(f_regas)) {
    [...]
}
```

**Excerto 2: listar_plantas()**

*Preste atenção à condição de paragem do ciclo.*

```
void listar_plantas() {
    printf("=== PLANTAS ===\n");
    for (int i = 0; i <= total_plantas; i++) {
        printf("ID: %d | Nome: %s | [...]\n", plantas[i].id, plantas[i].nome,
[...]);
    }
}
```

**Excerto 3: registar_rega()**

*Observe o que é verificado (e o que não é) antes de escrever nos arrays.*

```
int registar_rega(int id_planta, int data, int quantidade) {
    regas[total_regas].id_rega       = total_regas + 1;
    regas[total_regas].id_planta     = id_planta;
    regas[total_regas].data_rega     = data;
    regas[total_regas].quantidade_agua = quantidade;
    total_regas++;
    [...]
    return 1;
}
```

**Excerto 4: verificar_rega()**

*Analise o que a função devolve e pense em como a poderia usar noutro contexto do programa.*

```
void verificar_rega(int data_atual) {
    printf("=== PLANTAS QUE PRECISAM DE REGA ===\n");
    for (int i = 0; i < total_plantas; i++) {
        int 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);
        }
    }
}
```

**Excerto 5: criar_tarefa()**

*Observe o que acontece quando total_tarefas se aproxima do limite.*

```
void criar_tarefa(char* descricao, int data_prevista) {
    tarefas[total_tarefas].id_tarefa    = total_tarefas + 1;
    strcpy(tarefas[total_tarefas].descricao, descricao);
    tarefas[total_tarefas].data_prevista = data_prevista;
    tarefas[total_tarefas].concluida     = 0;
    total_tarefas++;
}
```

**Excerto 6: listar_tarefas_pendentes()**

*Analise a condição dentro do if. Execute mentalmente a função para ver o que produz.*

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

**Excerto 7: concluir_tarefa()**

*Analise a condição dentro do if. O que acontece quando id é, por exemplo, 3?*

```
void concluir_tarefa(int id) {
    for (int i = 0; i < total_tarefas; i++) {
        if (tarefas[i].id_tarefa = id) {
            tarefas[i].concluida = 1;
            [...]
            return;
        }
    }
    [...]
}
```

**Excerto 8: guardar_dados()**

*O que é guardado? O que fica por guardar? Pense no que acontece ao reiniciar o programa.*

```
void guardar_dados() {
    FILE *f_plantas = fopen("plantas.csv", "w");
    if (f_plantas == NULL) { printf("Erro\n"); return; }
    for (int i = 0; i < total_plantas; i++) {
        fprintf(f_plantas, "%d,%s,%s,%s,%d,%d\n", plantas[i].id, [...]);
    }
    fclose(f_plantas);
}
```

**Excerto 9: adicionar_planta()**

*A planta é adicionada ao array. O que acontece a essa informação quando o programa termina?*

```
void adicionar_planta(int id, char* nome, char* especie,
                      char* data_plantio, int intervalo) {
    [...]
    plantas[total_plantas].intervalo_rega = intervalo;
    plantas[total_plantas].ultima_rega    = 0; /* data atual do sistema */
    total_plantas++;
}
```

**Excerto 10: main() (leitura de *input*)**

*Observe como o input de texto é lido. O que acontece se o utilizador escrever um nome com espaços? O programa deve lidar corretamente com input textual (incluindo espaços).*

```
int main() {
    [...]
    switch (opcao) {
        [...]
        case 2: {
            int id, intervalo;
            char nome[50], especie[50], data[11];
            printf("Nome: ");    scanf("%s", nome);
            printf("Especie: "); scanf("%s", especie);
            [...]
            break;
        }
        [...]
        case 6: {
            char descricao[100]; int data_prevista;
            printf("Descricao: "); scanf("%s", descricao);
            [...]
            break;
        }
        [...]
    } while (opcao != 8);
    return 0;
}
```

## Tarefas a desenvolver

### Tarefa 1: Análise e correção de erros (1,5 valores)

a) Identifique todos os erros presentes nos excertos fornecidos. Não serão considerados como erros relevantes aspetos meramente estilísticos (ex: nomes de variáveis), exceto quando afetem gravemente a legibilidade ou compreensão do programa.

b) Para cada erro, explique:
   - A categoria (ex.: lógica incorreta, violação de pré-condição, gestão de ficheiros, leitura de *input*, etc);
   - As consequências em execução: o que acontece de errado e em que circunstâncias;
   - A correção aplicada e a justificação da escolha.

c) Apresente o código corrigido (em anexo dentro do relatório).

**Nota:** o enunciado não indica quantos erros existem. Encontrá-los todos, incluindo os menos óbvios, é parte essencial da avaliação. Erros redundantes ou repetidos devem ser agrupados e analisados em conjunto.

Na criação de novos itens, os IDs devem ser geridos automaticamente pelo sistema, sem necessidade de *input* do utilizador.

Após registar uma rega, o campo ultima_rega da planta correspondente deve ser atualizado para a data da rega.

## Tarefa 2: Reorganização modular (1,5 valores)

a) Proponha e implemente uma arquitetura modular para o projeto, onde o código final deve integrar as correções efetuadas. **A divisão em módulos é uma decisão sua,** o que importa é que seja correta, coerente e justificada. O código completo (incluindo as partes omitidas nos excertos) é inteiramente escrito por si.

b) Para cada módulo criado, indique:
   - Quais as funções públicas (expostas no .h) e quais as privadas (static no .c);
   - Que estruturas de dados utilizou e porquê;
   - Como garantiu o encapsulamento.

c) Explique as suas decisões:
   - Porque dividiu desta forma? Que alternativas considerou e rejeitou? Deve indicar pelo menos uma alternativa plausível à solução adotada e justificar a opção escolhida.
   - Que vantagens traz esta organização?
   - Como ficou a gestão de dependências entre módulos?

**Nota:** Desenvolva um menu para aceder às funcionalidades do programa.

Os módulos devem ser organizados de forma que nenhum módulo dependa *diretamente* dos *arrays* globais originais (plantas, regas, tarefas). O acesso a esses dados deve ser feito através de funções de módulo específico.

Atenção: a implementação do encapsulamento (módulos sem dependência direta dos *arrays* globais) é um dos aspetos mais exigentes deste trabalho.

## Tarefa 3: Relatório final (1,0 valor)

Elabore um relatório (máx. 10 páginas, exceto capa e código desenvolvido) com a seguinte estrutura:

- Introdução: objetivo do trabalho e abordagem adotada;
- Tarefa 1: Erros encontrados: tabela com erros, categoria, consequências e correções;
- Tarefa 2: Organização modular: descrição dos módulos, *interfaces* e decisões;
- Conclusão: desafios enfrentados, aprendizagens e possíveis melhorias futuras;
- Código desenvolvido: código completo corrigido e modularizado, em formato de texto.

Cada erro identificado deve estar explicitamente associado à respetiva correção no código (ex: referência à função/trecho alterado).

As justificações devem ser técnicas e concretas, evitando descrições genéricas (ex: “melhor organização”). Deve indicar o impacto da alteração no comportamento ou estrutura do programa.

O relatório deve ser conciso, mas suficientemente detalhado para demonstrar compreensão das alterações efetuadas.

O código e o relatório devem ser consistentes entre si. Incoerências entre o que é descrito e o que é implementado serão penalizadas.

**Nota**: O código desenvolvido deve ser incluído integralmente no relatório em formato de texto, para efeitos de avaliação.

## Critérios de avaliação e cotação (total 4 valores)

A avaliação do presente E-Fólio baseia-se numa apreciação global da qualidade do trabalho em cada critério. Soluções que não permitam executar corretamente os requisitos funcionais mínimos serão penalizadas, mesmo com boa análise teórica.

A pontuação atribuída em cada componente não resulta da contagem direta de elementos, mas da avaliação do desempenho global do estudante face aos objetivos definidos.

| Critério | Componentes | Cotação |
|---|---|---|
| Correção de erros | Avalia a capacidade de:<br>• Identificar erros relevantes nos excertos fornecidos;<br>• Compreender a sua natureza;<br>• Explicar consequências;<br>• Propor correções adequadas e justificadas.<br>**Níveis de desempenho:**<br>• **Insuficiente (0,0 - 0,4)**: Não identifica erros relevantes ou apresenta correções incorretas.<br>• **Básico (0,5 - 0,9)**: Identifica erros mais evidentes, com explicações limitadas e correções parciais.<br>• **Bom (1,0 - 1,2)**: Identifica um conjunto significativo de erros, com correções geralmente adequadas e alguma justificação.<br>• **Muito Bom / Excelente (1,3 - 1,5)**: Identificação abrangente (incluindo erros não triviais), com explicações claras, corretas e bem fundamentadas. | 1,5 |
| Modularização | Avalia a qualidade da organização do sistema, nomeadamente:<br>• Estrutura modular proposta;<br>• Separação de responsabilidades;<br>• Definição de *interfaces*;<br>• Encapsulamento de dados;<br>• Justificação das decisões adotadas.<br>**Níveis de desempenho:**<br>• **Insuficiente (0,0 - 0,4)**: Código monolítico ou sem estrutura coerente.<br>• **Básico (0,5 - 0,9)**: Estrutura limitada, com organização pouco consistente.<br>• **Bom (1,0 - 1,2)**: Modularização coerente, com separação funcional adequada e alguma justificação.<br>• **Muito Bom / Excelente (1,3 - 1,5)**: Estrutura clara, consistente e bem organizada, com boas decisões de *design* e justificações fundamentadas.<br>Como requisito mínimo para obter um nível básico, o código demonstra encapsulamento adequado, com módulos que não acedem diretamente aos *arrays* globais, utilizando funções de acesso adequadas. | 1,5 |
| Relatório | Avalia a qualidade da comunicação técnica, incluindo:<br>• Clareza e organização;<br>• Estrutura do documento;<br>• Qualidade da análise apresentada;<br>• Cumprimento das orientações formais.<br>**Níveis de desempenho:**<br>• **Insuficiente (0,0 - 0,2)**: Relatório inexistente, desorganizado ou irrelevante.<br>• **Básico (0,3 - 0,5)**: Estrutura limitada, com explicações superficiais.<br>• **Bom (0,6 - 0,8)**: Relatório claro e estruturado, com análise adequada.<br>• **Muito Bom / Excelente (0,9 - 1,0)**: Relatório bem estruturado, claro e com análise aprofundada e bem fundamentada. | 1,0 |

A classificação em cada critério resulta de uma apreciação global do desempenho, podendo situar-se em qualquer valor dentro dos intervalos definidos.

## Normas a respeitar

- O código tem de compilar de modo a poder ser avaliado (Visual Studio Code, com o *software* MinGW (Minimalist GNU for Windows), que inclui o compilador gcc);
- Deve entregar dois ficheiros:
  - relatorio_NNNNNN_efolioA.pdf (O relatório deve incluir, obrigatoriamente, todo o código fonte desenvolvido em formato de texto (não são aceites referências externas ao código);
  - codigo_NNNNNN_efolioA.zip (Deve conter o código fonte organizado e o executável para Windows (.exe), permitindo a execução do programa. Pode comprimir e entregar toda a pasta do projeto);
  - Ambos os ficheiros são de entrega obrigatória. Trabalhos sem código incluído no PDF ou sem executável no ficheiro zip não poderão ser corretamente avaliados.
- Nomeie os ficheiros com o seu número de estudante, seguido da identificação do E-Fólio, segundo o exemplo apresentado: relatorio_123456_efolioA.pdf;
- Deve carregar os referidos ficheiros para a plataforma no dispositivo E-Fólio A no seu espaço de turma, até à data e hora limite de entrega;
- Evite a entrega próximo da hora limite;
- Não serão aceites ficheiros fora do prazo, nem por outros meios, nomeadamente correio eletrónico;
- Não serão aceites trabalhos com indícios de plágio.

## Diretrizes para o texto do relatório

- Fonte: Arial ou Times New Roman, tamanho 11
- Espaçamento: 1,5 entre linhas
- Margens: 2,5 cm em todos os lados
- Formato: PDF (preferencial) ou DOCX
- Máximo: 10 páginas (excluindo capa e código desenvolvido). Embora a capacidade de síntese seja relevante e de certo modo fator de avaliação, o limite descrito é uma referência para que não hajam trabalhos muito curtos ou muito extensos. Neste sentido, se tiver pouco menos ou pouco mais texto do que indicado, não se aplicarão penalizações.

## Votos de bom trabalho!