

# C para quem conhece Java

A stylized icon of a computer monitor with a glowing blue border, centered around the code block.

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

int x;



Luís Simões da Cunha (2025)

# Índice

✳️ Parte 1 – O Essencial do C: Da Escrita à Compilação .....	5
⌚ O Ciclo de Vida de um Programa em C .....	5
🧠 O teu primeiro programa em C .....	5
💬 Comentários em C .....	5
📝 Tipos de dados básicos .....	6
➕ Operadores básicos .....	6
🆚 Diferenças culturais: Java vs C .....	7
✓ Exercício para ti .....	7
🧠 O Essencial para Memorizar .....	7
⌚ Parte 2 – Controle de Fluxo: if, else, switch e operadores lógicos .....	8
🧠 A Lógica da Tomada de Decisão .....	8
🔗 Operadores Lógicos e de Comparação .....	8
🧭 Condições Compostas .....	9
🧱 Estrutura switch .....	9
✓ Exemplo Prático .....	10
🧠 O Essencial para Memorizar .....	10
⌚ Parte 3 – Ciclos com for, while e do...while .....	11
⌚ 1. O ciclo while .....	11
⌚ 2. O ciclo do...while .....	11
⌚ 3. O ciclo for .....	11
📌 break e continue .....	12
🧠 Diferenças-chave entre os ciclos .....	12
✓ Exemplo completo .....	13
🧠 Boas práticas .....	13
🧠 Essencial para Memorizar .....	14
🔧 Parte 4 – Funções e o Fluxo da Execução .....	15
◆ 1. Criar Funções em C .....	15
📊 Tipos de retorno: void, int, float, char .....	15
⌚ 2. Passagem por Valor vs. por Referência .....	16
📌 3. Declarações e Protótipos de Funções .....	16
✓ Exemplo completo com tudo: .....	17

	Essencial para Memorizar .....	18
	Parte 5 – Arrays e Strings .....	19
	1. Declaração e Manipulação de Arrays.....	19
	2. Strings como Arrays de char .....	19
	3. Funções úteis da <code>string.h</code> .....	20
	Dicas Importantes .....	20
	Resumo do Essencial.....	21
	Parte 6 – Ponteiros com Clareza: Memória, Referência e Arrays.....	22
	1. Conceito de Ponteiro: o que são * e & .....	22
	2. Arrays vs Ponteiros .....	22
	3. O operador ->: ponteiros para estruturas .....	23
	4. Armadilhas comuns .....	23
	Exemplo completo .....	23
	Essencial para Memorizar .....	24
	Parte 7 – Estruturas (struct) e Organização de Dados .....	25
◆	1. Criar e Usar <code>struct</code> .....	25
	2. Arrays de Estruturas .....	25
	3. Ponteiros para Estruturas .....	26
	4. Comparações com Classes Simples em Java .....	26
	Exemplo Completo .....	27
	Essencial para Memorizar .....	27
	Parte 8 – Ficheiros e Entrada/Saída em C .....	28
	1. Abrir Ficheiros: <code>fopen()</code> .....	28
	2. Leitura e Escrita em Ficheiros de Texto .....	28
	3. Ficheiros Binários: <code>fread()</code> e <code>fwrite()</code> .....	29
	4. Buffering e Fecho com <code>fclose()</code> .....	29
	5. Erros Comuns .....	29
	Exemplo Completo .....	29
	Essencial para Memorizar .....	30
	Parte 9 – O Poder dos Modificadores: <code>extern</code> , <code>static</code> , <code>register</code> , <code>volatile</code> .....	31
◆	1. <code>extern</code> – Visibilidade Global entre Ficheiros .....	31
◆	2. <code>static</code> – Persistência ou Restrição de Acesso .....	31

⚡ 3. register – Guarda variável no registo do processador .....	31
⌚ 4. volatile – Nunca assumes que o valor não muda “sozinho” .....	32
🧠 Comparações com Java .....	32
✅ Exemplo completo e comentado .....	32
🧠 Essencial para Memorizar .....	33
📝 Parte 10 – Gestão de Memória e Boas Práticas .....	34
🧠 1. Alocação Dinâmica: malloc(), calloc(), realloc(), free().....	34
⚠ 2. Boas Práticas: evitar fugas de memória .....	34
💡 3. Estilo, Indentação e Modularização .....	35
✅ Essencial para Memorizar .....	36

## Parte 1 – O Essencial do C: Da Escrita à Compilação

### O Ciclo de Vida de um Programa em C

Em C, o processo de desenvolvimento segue 5 passos fundamentais:

1.  **Escreves o código-fonte** num ficheiro `.c` usando um editor (como o Code::Blocks ou VS Code).
2.  **Compilas o código** com um *compilador C* (como `gcc` ou o embutido no IDE).
3.  **Depuras (debug)** os erros de compilação (sintaxe, tipos, etc.).
4.  **Executas** o programa compilado (geralmente um ficheiro `.exe` ou `.out`).
5.  **Iteras:** corriges e testas até o comportamento ser o desejado.

 Em Java, a compilação transforma `.java` em `.class`, e a JVM interpreta ou executa esse bytecode. Em C, o compilador gera código nativo diretamente.

---

### O teu primeiro programa em C

```
#include <stdio.h>
```

```
int main() {
    printf("Olá, mundo em C!\n");
    return 0;
}
```

 Vamos analisar:

Elemento	Explicação
<code>#include &lt;stdio.h&gt;</code>	Instrução ao pré-processador para incluir o cabeçalho de I/O
<code>int main()</code>	Função principal (ponto de entrada do programa)
<code>{ ... }</code>	Bloco de código delimitado por chaves
<code>printf(...)</code>	Função que escreve no ecrã (parecido com <code>System.out.print</code> )
<code>\n</code>	Caractere de nova linha (newline)
<code>return 0;</code>	Indica que o programa terminou com sucesso

 **Nota:** Ao contrário de Java, não há `class`, `public` nem `static` aqui. C é mais próximo do "metal".

---

### Comentários em C

Dois estilos:

```
/* Comentário de várias linhas */  
  
// Comentário de uma linha (como em C++ ou Java)
```

 Usa comentários generosamente para manter o código legível (especialmente útil quando voltares semanas depois!).

---

## Tipos de dados básicos

C é *fortemente tipado* (como Java), mas mais “manual”. Os principais tipos são:

Tipo em C	Equivalente em Java	Exemplo
int	int	int idade = 30;
float	float	float peso = 70.5;
double	double	double pi = 3.1415;
char	char	char letra = 'A';

**Declaração de variáveis:**

```
int idade = 25;  
float preco = 12.5;  
char inicial = 'L';
```

 *Dica de memória:* char ocupa 1 byte, int costuma ocupar 4 bytes (mas depende da máquina!), e float ou double podem ocupar 4 ou 8 bytes, respetivamente.

---

## Operadores básicos

São praticamente os mesmos do Java:

```
+ - * / %      // Soma, subtração, multiplicação, divisão, resto  
= == != < >      // Atribuição e comparação  
  
int a = 5, b = 2;  
printf("Soma: %d\n", a + b);      // Soma: 7  
printf("Divisão: %d\n", a / b); // Divisão: 2 (sem casas decimais!)
```

 Em C, divisão de inteiros devolve inteiro! Usa float se quiseres resultado com vírgulas.

## Diferenças culturais: Java vs C

Conceito	Java	C
Ambiente de execução	JVM (bytecode)	Código nativo
Organização em classes	Obrigatório	Inexistente
Função principal	<code>public static void main</code>	<code>int main()</code>
Gestão de memória	Automática (GC)	Manual ( <code>malloc/free</code> )
Bibliotecas padrão	Muito extensas	Mais simples e enxutas

---

## Exercício para ti

Cria e executa este programa:

```
#include <stdio.h>

int main() {
    int idade = 15;
    float altura = 1.75;
    char inicial = 'L';

    printf("Idade: %d\n", idade);
    printf("Altura: %.2f metros\n", altura);
    printf("Inicial do nome: %c\n", inicial);

    return 0;
}
```

---

## O Essencial para Memorizar

- Todos os programas em C têm uma função `main()`.
  - Usa `#include <stdio.h>` para poderes usar `printf()`.
  - Declara variáveis com tipos explícitos (`int`, `float`, etc.).
  - `\n` serve para quebrar linha no output.
  - Comentários podem ser `/* ... */` ou `// ...`
  - O compilador transforma `.c` em código executável.
-

## Parte 2 – Controle de Fluxo: if, else, switch e operadores lógicos

Nesta parte vais relembrar como tomar decisões em C, usando estruturas de controlo de fluxo que, embora familiares em Java, têm algumas diferenças subtils.

---

### A Lógica da Tomada de Decisão

Tal como em Java, um programa em C pode seguir diferentes caminhos consoante certas condições sejam verdadeiras ou falsas. A estrutura mais básica é o `if`:

```
int idade = 18;

if (idade >= 18) {
    printf("És maior de idade.\n");
} else {
    printf("Ainda és menor.\n");
}
```

◊ *Sintaxe geral do if:*

```
if (condição) {
    // código se a condição for verdadeira
} else {
    // código se a condição for falsa
}
```

---

### Operadores Lógicos e de Comparação

Estes operadores funcionam como em Java:

Operador	Significado	Exemplo
<code>==</code>	Igual a	<code>x == 10</code>
<code>!=</code>	Diferente de	<code>x != 5</code>
<code>&lt;</code>	Menor que	<code>x &lt; 7</code>
<code>&gt;</code>	Maior que	<code>x &gt; 3</code>
<code>&lt;=</code>	Menor ou igual a	<code>x &lt;= 10</code>
<code>&gt;=</code>	Maior ou igual a	<code>x &gt;= 0</code>
<code>&amp;&amp;</code>	E lógico ( <i>and</i> )	<code>x &gt; 0 &amp;&amp; y &gt; 0</code>
<code>'</code>		<code>'</code>
<code>!</code>	Negação lógica	<code>!(x &gt; 10)</code>

## Lembra-te:

- `=` é **atribuição**
  - `==` é **comparação**
- 

## Condições Compostas

```
int idade = 20;
int temCarta = 1; // 1 = true

if (idade >= 18 && temCarta) {
    printf("Podes conduzir.\n");
}
```

 Em C, qualquer valor diferente de zero é considerado **verdadeiro**.

---

## Estrutura switch

Ideal para testar uma variável contra múltiplos valores. Evita múltiplos `if...else if`.

```
int opcao = 2;

switch (opcao) {
    case 1:
        printf("Selecionaste 1.\n");
        break;
    case 2:
        printf("Selecionaste 2.\n");
        break;
    case 3:
        printf("Selecionaste 3.\n");
        break;
    default:
        printf("Opção inválida.\n");
}
```

### Regras importantes:

- `break` impede que o código “caia” para o próximo `case`.
- `default` é opcional mas recomendado — age como o `else`.

 Em Java, podes usar `String` ou `enum` no `switch`, mas em C é só para `int` ou `char`.

---

## Exemplo Prático

```
#include <stdio.h>

int main() {
    int nota = 16;

    if (nota >= 18) {
        printf("Excelente!\n");
    } else if (nota >= 14) {
        printf("Bom!\n");
    } else if (nota >= 10) {
        printf("Suficiente.\n");
    } else {
        printf("Reprovado.\n");
    }

    return 0;
}
```

---

## O Essencial para Memorizar

- Usa `if`, `else if`, `else` para lógica condicional simples ou composta.
- Os operadores `&&`, `||` e `!` permitem combinar ou negar condições.
- A estrutura `switch` é mais legível que muitos `if...else if` em situações com múltiplas escolhas.
- Em C, **qualquer número diferente de zero é verdadeiro**.

## Parte 3 – Ciclos com `for`, `while` e `do...while`

Nesta parte, vamos relembrar os **3 tipos principais de laços em C**, comparar as suas diferenças e ver como usar `break`, `continue` e boas práticas para tornar os teus ciclos claros e eficientes — sempre com paralelos ao que já conheces em Java.

---

### 1. O ciclo `while`

Usado quando **não sabes ao certo quantas vezes o ciclo vai repetir** e queres testar a condição **antes** da primeira execução.

```
int i = 0;
while (i < 5) {
    printf("Valor de i: %d\n", i);
    i++;
}
```

 *Em Java seria igual: `while (i < 5) {...}`.*

---

### 2. O ciclo `do...while`

Executa o corpo **pelo menos uma vez**, e **só depois** testa a condição:

```
int i = 0;
do {
    printf("Executou com i = %d\n", i);
    i++;
} while (i < 5);
```

 Ideal para menus ou situações onde precisas de garantir **uma execução mínima**.

---

### 3. O ciclo `for`

Perfeito para contagens controladas — quando **sabes de antemão** quantas vezes vais repetir.

```
for (int i = 0; i < 5; i++) {
    printf("i vale: %d\n", i);
}
```

 Este ciclo tem 3 partes:

- **inicialização** → `int i = 0`

- **condição** →  $i < 5$
- **atualização** →  $i++$

💡 *Igualzinho ao Java!*

---

### 📝 break e continue

Estas palavras-chave dão-te **controlo total** sobre o fluxo dos ciclos:

◊ **break** → *Sai imediatamente do ciclo*

```
for (int i = 0; i < 10; i++) {  
    if (i == 5) break;  
    printf("%d ", i);  
}  
// Saída: 0 1 2 3 4
```

◊ **continue** → *Salta o resto do corpo e volta à condição*

```
for (int i = 0; i < 5; i++) {  
    if (i == 2) continue;  
    printf("%d ", i);  
}  
// Saída: 0 1 3 4
```

---

### ⌚ Diferenças-chave entre os ciclos

Tipo	Quando usar	Testa condição	Executa pelo menos 1 vez?
while	Quando não sabes quantas vezes vais repetir	Antes	Não
do...while	Quando precisas de 1 execução garantida	Depois	Sim
for	Quando sabes o número de repetições	Antes	Não (mas estrutura mais concisa)

---

## Exemplo completo

```
#include <stdio.h>

int main() {
    int i;

    printf("Ciclo for:\n");
    for (i = 0; i < 3; i++) {
        printf("for i=%d\n", i);
    }

    printf("\nCiclo while:\n");
    i = 0;
    while (i < 3) {
        printf("while i=%d\n", i);
        i++;
    }

    printf("\nCiclo do...while:\n");
    i = 0;
    do {
        printf("do...while i=%d\n", i);
        i++;
    } while (i < 3);

    return 0;
}
```

---

## Boas práticas

- Evita ciclos infinitos (`while(1)`) sem `break` justificado.
- Usa nomes de variáveis que **expliquem o propósito** (`contador`, `tentativas`, etc.).
- Comenta ciclos complexos.
- Prefere `for` quando possível — mais legível para contagens simples.

---

## Essencial para Memorizar

- `while` → Testa primeiro, executa se for verdade.
  - `do...while` → Executa primeiro, testa depois.
  - `for` → Ideal para contagens.
  - `break` → Sai do ciclo.
  - `continue` → Salta para a próxima iteração.
  - C é idêntico ao Java nos ciclos, mas não há objetos nem coleções — tudo mais “nu e cru”.
-

## Parte 4 – Funções e o Fluxo da Execução

Nesta parte vais reaprender a declarar e usar **funções em C**, compreender o que significa passar variáveis **por valor ou por referência**, e aprender a usar **protótipos de funções** — tudo com explicações claras, exemplos e paralelos com o Java.

---

### ◊ 1. Criar Funções em C

Em C, **funções** são blocos de código reutilizáveis que executam uma tarefa. Cada função tem:

- um **tipo de retorno** (`int`, `float`, `void`, etc.)
- um **nome**
- **parênteses com os parâmetros (opcional)**
- um **bloco de código {}**

 *Exemplo simples:*

```
#include <stdio.h>

void dizOla() {
    printf("Olá do interior da função!\n");
}

int main() {
    dizOla(); // chama a função
    return 0;
}
```

 *Em Java, seria como um método `void dizOla()` dentro de uma classe.*

---

### Tipos de retorno: `void`, `int`, `float`, `char`

- `void` → a função **não retorna valor**
- `int` → retorna um **número inteiro**
- `float` → retorna um número com casas decimais
- `char` → retorna um **carácter**

 *Exemplo com retorno:*

```
int soma(int a, int b) {
    return a + b;
}

int main() {
```

```
int resultado = soma(3, 4);
printf("Resultado: %d\n", resultado);
return 0;
}
```

---

## ☒ 2. Passagem por Valor vs. por Referência

### ▢ Passagem por Valor

Por defeito, C **passa variáveis por valor** – ou seja, **faz uma cópia**. Alterar dentro da função **não afeta o original**:

```
void modifica(int x) {
    x = x + 10;
}

int main() {
    int a = 5;
    modifica(a);
    printf("a = %d\n", a); // continua 5
}
```

### ▢ Passagem por Referência (com ponteiros)

Para alterar o valor original, passas **o endereço de memória** usando **ponteiros** (\* e &):

```
void modifica(int *x) {
    *x = *x + 10;
}

int main() {
    int a = 5;
    modifica(&a); // passa o endereço
    printf("a = %d\n", a); // agora é 15
}
```

🧠 Em Java, só tens passagem por valor — mas os objetos são passados por valor da referência.

---

## ❖ 3. Declarações e Protótipos de Funções

Em C, as funções precisam de ser **declaradas antes de usadas**. Podes fazer isso com um **protótipo** no início do ficheiro:

```
int soma(int, int); // protótipo
```

```
int main() {
    printf("Soma: %d\n", soma(2, 3));
    return 0;
}

int soma(int a, int b) { // definição
    return a + b;
}
```

💡 Se não usares protótipo, tens de garantir que a função aparece **antes do main()**.

---

### Exemplo completo com tudo:

```
#include <stdio.h>

// protótipos
void mostraMensagem();
int multiplica(int, int);
void dobrar(int *);

int main() {
    mostraMensagem();
    int resultado = multiplica(3, 5);
    printf("Multiplicação: %d\n", resultado);

    int valor = 4;
    dobrar(&valor);
    printf("Valor dobrado: %d\n", valor);
    return 0;
}

void mostraMensagem() {
    printf("Funções são poderosas!\n");
}

int multiplica(int a, int b) {
    return a * b;
}

void dobrar(int *) {
    *n = *n * 2;
}
```

---

## Essencial para Memorizar

- Toda a função precisa de:
  - Tipo de retorno
  - Nome
  - Parâmetros (ou não)
  - Bloco {} com instruções
- **Por omissão**, C passa **por valor**.
- Usa **ponteiros** (\*) e (&) para modificar valores originais.
- Declara protótipos antes do `main()` para garantir que as chamadas são reconhecidas.

## Parte 5 – Arrays e Strings

Nesta parte vais aprender (ou reprender) como funcionam **arrays** e **strings** em C, e como tirar partido das funções da biblioteca `string.h` — sempre com exemplos práticos e paralelos a Java.

---

### 1. Declaração e Manipulação de Arrays

#### ◊ *O que é um array?*

Um **array** em C é uma coleção de elementos do mesmo tipo, armazenados **em posições consecutivas da memória**.

#### *Exemplo – Array de inteiros:*

```
int numeros[5] = {10, 20, 30, 40, 50};
```

- `int` → tipo de dados
- `numeros` → nome do array
- `[5]` → número de elementos
- `{...}` → valores iniciais

*Aceder aos elementos:*

```
printf("Primeiro número: %d\n", numeros[0]); // 10
numeros[2] = 99; // altera o terceiro valor
```

#### Os índices começam em **0**, como em Java!

#### *Exemplo com for:*

```
for (int i = 0; i < 5; i++) {
    printf("Elemento %d: %d\n", i, numeros[i]);
}
```

---

### 2. Strings como Arrays de char

Em C, **uma string é simplesmente um array de char** terminado por `\0` (carácter nulo).

#### ◊ *Declaração e inicialização:*

```
char nome[6] = {'L', 'u', 'i', 's', '\0'};
```

Mas normalmente usamos a forma mais prática:

```
char nome[] = "Luis";
```

 "Luis" é na verdade: `{'L', 'u', 'i', 's', '\0'}`

Aceder ou alterar caracteres:

```
printf("Primeira letra: %c\n", nome[0]); // L
nome[0] = 'R';
```

---

### 3. Funções úteis da string.h

Para usar funções de strings, tens de incluir:

```
#include <string.h>
```

◊ As mais úteis:

Função	O que faz	Exemplo
strlen(str)	Devolve o comprimento da string (sem \0)	strlen("Luis") → 5
strcpy(dest, src)	Copia src para dest	strcpy(nome2, nome1);
strcat(dest, src)	Junta src no fim de dest	strcat(nome, " Silva");
strcmp(str1, str2)	Compara strings (0 = iguais)	if (strcmp(a,b) == 0)

 Exemplo completo:

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

int main() {
    char nome[50] = "Luis";
    char apelido[] = " Cunha";

    strcat(nome, apelido);
    printf("Nome completo: %s\n", nome);
    printf("Tamanho: %lu\n", strlen(nome));

    return 0;
}
```

---

### Dicas Importantes

-  **NÃO** declares `char nome[5] = "Luis";` → falta espaço para o \0.
  -  **strcpy e strcat não protegem contra overflow de memória** — usa `strncpy`, `strncat` se quiseres mais segurança.
  - As strings em C **não são objetos**, são apenas **arrays com \0 no fim**.
-

## Resumo do Essencial

- Arrays são coleções de dados do mesmo tipo, acessíveis por índice.
- Strings em C são arrays de `char` com `\0` no fim.
- Usa `string.h` para operações como copiar, comparar e medir strings.
- Não há verificação automática de limites como em Java — cuidado com o tamanho dos arrays!

## Parte 6 – Ponteiros com Clareza: Memória, Referência e Arrays

Os **ponteiros** são o coração da linguagem C. Embora possam parecer assustadores ao início, tornam-se rapidamente teus aliados quando entedes a lógica por trás: **endereços de memória e acesso indireto a dados**. Esta parte vai clarificar os conceitos de \*, &, ->, [ ], e alertar para armadilhas comuns.

---

### 1. Conceito de Ponteiro: o que são \* e &

◊ *Um ponteiro é uma variável que guarda o endereço de outra variável.*

```
int a = 10;  
int *p;      // declara um ponteiro para int  
p = &a;      // p guarda o endereço de a
```

Símbolo	Significado
&a	"endereço de a"
*p	"valor guardado no endereço apontado por p"

```
printf("%d\n", *p); // imprime 10
```

 Ou seja:

- & → operador de **endereço**
  - \* → operador de **desreferência** (acesso indireto)
- 

### 2. Arrays vs Ponteiros

Em C, **arrays e ponteiros estão intimamente ligados**:

```
int valores[3] = {10, 20, 30};  
int *ptr = valores;
```

 **valores** é o endereço do 1.º elemento → equivalente a **&valores[0]**

Podes usar **ptr[i]**, **\*(ptr + i)** ou **valores[i]** de forma equivalente:

```
printf("%d\n", ptr[1]);      // 20  
printf("%d\n", *(valores+2)); // 30
```

 **Nota importante:** arrays têm **tamanho fixo**, enquanto ponteiros **podem apontar para qualquer localização** (incluindo memória alocada dinamicamente).

---

## ⚡ 3. O operador ->: ponteiros para estruturas

Se tiveres um **ponteiro para estrutura**, podes usar `->` como atalho para `(*p).campo`.

```
struct Pessoa {  
    char nome[20];  
    int idade;  
};  
  
struct Pessoa p = {"Luis", 45};  
struct Pessoa *ptr = &p;  
  
printf("%s tem %d anos\n", ptr->nome, ptr->idade);
```

🧠 `ptr->idade` é igual a `(*ptr).idade`.

---

## ⚠ 4. Armadilhas comuns

### ✗ Acesso fora de limites

```
int a[3] = {1, 2, 3};  
printf("%d\n", a[5]); // comportamento indefinido!
```

⚠ O compilador **não impede** isto! A memória acedida pode conter lixo ou causar falhas.

### ✗ Ponteiros não inicializados

```
int *p;  
*p = 42; // ERRO! p não aponta para nada válido
```

Sempre inicializa os ponteiros antes de os usar!

### ✓ Solução:

```
int x = 10;  
int *p = &x;  
*p = 42; // agora é válido
```

---

### ✓ Exemplo completo

```
#include <stdio.h>  
  
int main() {  
    int valor = 50;  
    int *ptr = &valor;  
  
    printf("Valor: %d\n", valor);
```

```
printf("Endereço: %p\n", (void*)ptr);
printf("Valor via ponteiro: %d\n", *ptr);

int numeros[3] = {10, 20, 30};
printf("numeros[1] = %d\n", *(numeros + 1)); // 20

return 0;
}
```

---

## Essencial para Memorizar

- `*` → desreferência (acede ao valor apontado)
- `&` → endereço de uma variável
- `[]` → açúcar sintático para `*(p + i)`
- `->` → acede a membros de estruturas através de ponteiro
- Arrays e ponteiros estão fortemente relacionados
-  Evita ultrapassar os limites dos arrays ou usar ponteiros não inicializados

## Parte 7 – Estruturas (`struct`) e Organização de Dados

As `struct` são a forma mais poderosa e flexível de **organizar dados compostos** em C — algo próximo das classes simples em Java, mas sem métodos ou encapsulamento. Vamos explorar como criar, usar e manipular estruturas, incluindo **arrays de struct** e **ponteiros para estruturas**.

---

### ◊ 1. Criar e Usar `struct`

Uma `struct` agrupa vários tipos de dados sob um mesmo nome — por exemplo, para representar uma pessoa:

```
struct Pessoa {  
    char nome[50];  
    int idade;  
    float altura;  
};
```

Depois, podes declarar variáveis do tipo `Pessoa`:

```
struct Pessoa p1;  
  
strcpy(p1.nome, "Luis");  
p1.idade = 45;  
p1.altura = 1.75;
```

 *Isto seria equivalente a uma classe com três atributos públicos em Java.*

---

### 2. Arrays de Estruturas

Tal como podes ter arrays de `int`, também podes ter arrays de `struct`:

```
struct Pessoa turma[3];  
  
strcpy(turma[0].nome, "Ana");  
turma[0].idade = 20;  
turma[0].altura = 1.60;
```

 *Percorrer um array de structs:*

```
for (int i = 0; i < 3; i++) {  
    printf("Nome: %s | Idade: %d\n", turma[i].nome, turma[i].idade);  
}
```

 Cada `turma[i]` é uma estrutura com campos acessíveis com ponto (.).

---

### ⌚ 3. Ponteiros para Estruturas

Assim como os ponteiros para variáveis simples, também podes ter ponteiros para estruturas:

```
struct Pessoa p = {"João", 33, 1.80};  
struct Pessoa *ptr = &p;  
  
printf("%s tem %d anos\n", ptr->nome, ptr->idade);
```

#### 💡 Notas:

- `ptr->campo` é atalho para `(*ptr).campo`
  - Este acesso é comum em funções e em estruturas ligadas (como listas ligadas)
- 

### 🧠 4. Comparações com Classes Simples em Java

Conceito	Em C ( <code>struct</code> )	Em Java ( <code>class</code> )
Definição	<code>struct Pessoa {...};</code>	<code>class Pessoa { ... }</code>
Instância	<code>struct Pessoa p;</code>	<code>Pessoa p = new Pessoa();</code>
Acesso a campos	<code>p.nome, p.idade</code>	<code>p.nome, p.getIdade()</code>
Métodos	✗ (não há)	✓ com <code>void</code> e retorno
Encapsulamento	✗ tudo é público por omissão	✓ com <code>private</code> e <code>get/set</code>
Construtores	✗ (usa-se atribuição manual)	✓ com <code>new</code>

🗣 Em C, `struct` é puramente **dados**. Em Java, `class` tem dados + comportamento.

---

## Exemplo Completo

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

struct Livro {
    char titulo[50];
    int ano;
};

int main() {
    struct Livro l1 = {"O Alquimista", 1988};
    struct Livro *ptr = &l1;

    printf("Título: %s\n", ptr->titulo);
    printf("Ano: %d\n", ptr->ano);

    return 0;
}
```

---

## Essencial para Memorizar

- `struct` define um novo tipo que agrupa vários campos.
- Usa `.` para aceder a campos, e `->` quando usas ponteiros.
- Arrays de estruturas facilitam a organização de coleções de dados.
- `struct` ≈ classe simples em Java, mas **sem métodos nem encapsulamento**.

## Parte 8 – Ficheiros e Entrada/Saída em C

Trabalhar com ficheiros (textuais e binários) é essencial em C — quer para guardar resultados de programas, quer para manipular dados persistentes. Esta parte cobre os fundamentos de como abrir, ler, escrever e fechar ficheiros, com atenção ao buffering e aos erros comuns.

---

### 1. Abrir Ficheiros: `fopen()`

Para aceder a um ficheiro em C, usas a função `fopen()`:

```
FILE *ficheiro = fopen("dados.txt", "r");
```

- "r" → modo **leitura**
- "w" → **escreve** (apaga conteúdo anterior)
- "a" → **acrescenta** ao fim
- "rb", "wb", "ab" → modos **binários**

 *Verificação obrigatória:*

```
if (ficheiro == NULL) {
    printf("Erro ao abrir o ficheiro!\n");
    return 1;
}
```

---

### 2. Leitura e Escrita em Ficheiros de Texto

◊ *Escrever com `fprintf()`:*

```
FILE *f = fopen("saida.txt", "w");
fprintf(f, "Nome: %s | Idade: %d\n", "Luis", 45);
fclose(f);
```

◊ *Ler com `fscanf()`:*

```
FILE *f = fopen("entrada.txt", "r");
char nome[50];
int idade;

fscanf(f, "%s %d", nome, &idade);
fclose(f);
```

 **fscanf** segue o mesmo formato de `scanf`, mas **lê do ficheiro**.

---

### 3. Ficheiros Binários: `fread()` e `fwrite()`

Quando queres guardar **dados sem formatação** (ex. arrays, estruturas), usas ficheiros binários:

```
FILE *fb = fopen("dados.bin", "wb");
fwrite(&variavel, sizeof(variavel), 1, fb);
fclose(fb);
```

E para ler:

```
FILE *fb = fopen("dados.bin", "rb");
fread(&variavel, sizeof(variavel), 1, fb);
fclose(fb);
```

 Ideal para **estruturas completas**, arrays e melhor desempenho.

---

### 4. Buffering e Fecho com `fclose()`

Sempre que abres um ficheiro com `fopen`, deves fechá-lo com:

```
fclose(f);
```

Isto:

- Liberta recursos do sistema.
  - Garante que dados pendentes **no buffer são realmente gravados no disco**.
- 

### 5. Erros Comuns

Erro	Solução
Esquecer <code>fclose()</code>	Pode perder dados (não são gravados no disco).
Não verificar <code>fopen()</code>	Pode tentar ler/escrever num NULL, crasha o programa.
Formato errado em <code>fscanf()</code>	Pode ler lixo ou causar comportamento indefinido.
Usar "w" em vez de "a"	Apaga ficheiro sempre que abres.

---

### Exemplo Completo

```
#include <stdio.h>

int main() {
    FILE *f = fopen("dados.txt", "w");
    if (f == NULL) {
        printf("Erro a abrir ficheiro!\n");
    }
```

```
    return 1;
}

fprintf(f, "Olá, mundo do C!\n");
fclose(f);

f = fopen("dados.txt", "r");
char buffer[100];
fgets(buffer, 100, f);
printf("Lido do ficheiro: %s", buffer);
fclose(f);

return 0;
}
```

---

## Essencial para Memorizar

- Usa `FILE * e fopen()` para abrir ficheiros.
- `fprintf / fscanf` para texto, `fwrite / fread` para binários.
- Fecha sempre os ficheiros com `fclose()`.
- Verifica sempre se `fopen` resultou em `NULL`.

## Parte 9 – O Poder dos Modificadores: `extern`, `static`, `register`, `volatile`

Estes modificadores controlam **como e onde** as variáveis são **armazenadas, acessadas e partilhadas entre ficheiros**. São como “rótulos” que dizes ao compilador para alterar o comportamento padrão de uma variável.

---

### ◊ 1. `extern` – Visibilidade Global entre Ficheiros

`extern` declara uma **variável global definida noutro ficheiro**.

```
// num1.c
int valor = 42;

// num2.c
extern int valor;
printf("%d\n", valor); // valor vem do ficheiro num1.c
```

 Útil em programas **modulares**, para evitar redefinir variáveis.

 Em Java, equivalentes seriam **variáveis public static** visíveis em várias classes.

---

### ◊ 2. `static` – Persistência ou Restrição de Acesso

*Em funções/variáveis globais:*

- Torna a função ou variável visível **só naquele ficheiro (escopo interno)**.

```
static int contador = 0; // não visível fora do ficheiro
```

*Em funções locais:*

- Preserva o valor da variável entre chamadas — **persistência**.

```
void exemplo() {
    static int x = 0;
    x++;
    printf("%d\n", x);
}
```

 A cada chamada, `x` mantém o seu valor anterior (como uma variável de instância em Java).

---

### ⚡ 3. `register` – Guarda variável no registo do processador

Sugere ao compilador que a variável seja **armazenada num registo** (para acesso mais rápido).

```
register int i;
```

**⚠ Apenas sugestão** — o compilador pode ignorar. E não podes usar &i (porque não tem endereço visível na RAM).

---

## ⌚ 4. volatile – Nunca assumes que o valor não muda “sozinho”

Diz ao compilador: “não faças otimizações com esta variável”, porque pode mudar fora do controlo do programa.

Ex: variáveis ligadas a hardware, interrupções ou threads.

`volatile int estadoSensor;`

💡 Sem `volatile`, o compilador pode “guardar em cache” o valor e **não ver as alterações feitas por fora!**

---

## 🌐 Comparações com Java

Conceito	C	Java
Visibilidade externa	<code>extern</code>	<code>public static</code>
Persistência local	<code>static</code> (em função)	<code>static</code> ou variáveis de instância
Otimização CPU	<code>register</code>	O compilador Java gera automaticamente
Concorrência	<code>volatile</code> (informar o compilador)	<code>volatile</code> em Java também existe

---

## ☑ Exemplo completo e comentado

```
#include <stdio.h>

// Variável visível só neste ficheiro
static int totalChamadas = 0;

void conta() {
    static int chamadas = 0; // persiste entre chamadas
    chamadas++;
    totalChamadas++;
    printf("Chamadas locais: %d | Total: %d\n", chamadas, totalChamadas);
}

int main() {
    for (register int i = 0; i < 3; i++) {
        conta();
    }
}
```

```

volatile int sensor = 1; // simula leitura constante
printf("Sensor: %d\n", sensor);

return 0;
}

```

---

### Essencial para Memorizar

Modificador	Efeito Principal	Analogia Java
<code>extern</code>	Usa variável externa (noutro ficheiro)	<code>public static</code> entre classes
<code>static</code>	Mantém valor entre chamadas ou oculta visibilidade	<code>static</code> ou privado por classe
<code>register</code>	Otimiza para acesso rápido (CPU)	Controlado pelo compilador Java
<code>volatile</code>	Previne otimizações, valor pode mudar fora do programa	<code>volatile</code> em multithreading

## 🔗 Parte 10 – Gestão de Memória e Boas Práticas

Esta parte aborda o **controlo da memória dinâmica** em C com `malloc`, `calloc`, `realloc` e `free`, destacando também boas práticas para evitar fugas de memória e manter o código limpo e modular. Aqui é onde muitos programas “morrem lentamente” sem que o programador se aperceba. Vamos evitar isso com clareza, rigor e bons hábitos.

---

### ⌚ 1. Alocação Dinâmica: `malloc()`, `calloc()`, `realloc()`, `free()`

A memória dinâmica é usada quando **não sabes à partida o tamanho exato** dos dados que vais precisar.

#### 🌐 `malloc(size)`

Reserva um bloco de memória com `size` bytes.

```
int *v = (int*) malloc(5 * sizeof(int)); // 5 inteiros
```

- Retorna `void*` → por isso fazemos cast `((int*))`.
- A memória não é inicializada.

#### 🌐 `calloc(n, size)`

Reserva memória e inicializa tudo a zero.

```
int *v = (int*) calloc(5, sizeof(int)); // 5 inteiros a 0
```

#### 🌐 `realloc(ptr, new_size)`

Muda o tamanho de um bloco já alocado com `malloc` ou `calloc`.

```
v = realloc(v, 10 * sizeof(int));
```

⚠ Pode mudar o endereço! Guarda sempre o resultado.

#### 🌐 `free(ptr)`

Liberta a memória alocada dinamicamente.

```
free(v);
```

💡 Depois de `free`, o ponteiro fica “**pendurado**” – define-o como `NULL` por segurança.

---

### 🔔 2. Boas Práticas: evitar fugas de memória

#### 🌐 *Memory Leaks (fugas de memória)*

Ocorrem quando:

- Aloca memória mas nunca liberta (`malloc` sem `free`)
- Perdes a referência ao ponteiro antes de libertar

### Como evitar:

- Usa sempre `free` quando a memória já não for necessária
- Aponta o ponteiro para `NULL` depois de `free`
- Usa ferramentas como **Valgrind** (em Linux):

`valgrind --leak-check=full ./meu_programa`

 Valgrind deteta acessos inválidos e memória não libertada — como o “detetive” do C!

---

## 3. Estilo, Indentação e Modularização

Um bom programa C não é apenas funcional — é **legível, modular e fácil de manter**. Eis as boas práticas:

### Modularização

- Divide o código em **funções pequenas e coesas**
- Usa `*.h` para **headers** com declarações
- Usa `*.c` separados por funcionalidade

```
// ficheiro: calculadora.h
int soma(int a, int b);

// ficheiro: calculadora.c
int soma(int a, int b) { return a + b; }
```

### Nomeação e Estilo

- Nomes descritivos: `conta_alunos` em vez de `ca`
- Usa `snake_case` ou `camelCase` consistentemente
- Indenta com **4 espaços ou tab configurado**
- Coloca `{` na mesma linha da função (padrão K&R)

### Exemplo de boa prática:

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

int* criar_array(int n) {
    int* v = (int*) malloc(n * sizeof(int));
    if (v == NULL) {
        fprintf(stderr, "Erro de alocação\n");
        exit(1);
    }
    return v;
```

```

}

void libertar_array(int* v) {
    free(v);
    v = NULL;
}

int main() {
    int* dados = criar_array(10);

    for (int i = 0; i < 10; i++) dados[i] = i * 2;

    for (int i = 0; i < 10; i++) printf("%d ", dados[i]);

    libertar_array(dados);
    return 0;
}

```

---

### Essencial para Memorizar

Função	Uso Principal
<code>malloc</code>	Aloca memória sem inicializar
<code>calloc</code>	Aloca memória e inicializa com 0
<code>realloc</code>	Redimensiona bloco já alocado
<code>free</code>	Liberta memória

### Boas práticas:

- Sempre `free` depois de usar memória dinâmica
- Usa Valgrind para verificar fugas
- Modulariza o código em funções e ficheiros separados
- Mantém estilo limpo e indentado