Revisão Inicial — Arrays em C

> **Progresso (Arrays em C)**
>
> - ☑ Estudo inicial concluído
> - Revisões agendadas: (preencher)

1. Declaração e inicialização de arrays

- **Estáticos**: tamanho definido em tempo de compilação.

```
int arr[5] = {1, 2, 3, 4, 5};
int arr2[] = {10, 20, 30}; // tamanho inferido
```

- **Dinâmicos**: alocados em tempo de execução com `malloc` / `calloc`.

```
int *arr = malloc(n * sizeof(int));
if (!arr) { /* erro */ }
```

2. Ponteiros e aritmética de ponteiros

- Nome do array decai para `int*` quando passado para função.
- `*(arr + i)` é equivalente a `arr[i]`.

3. `sizeof` e decaimento

- Em escopo local: `sizeof(arr)` = tamanho total do array em bytes.
- Em função: `sizeof(param)` = tamanho do ponteiro (array decai para ponteiro).

4. Alocação dinâmica

```
int *arr = malloc(n * sizeof *arr);
arr = realloc(arr, novo_tam * sizeof *arr);
free(arr);
```

5. Arrays 2D

- **Contíguo** (um bloco único):

```
int *mat = malloc(linhas * colunas * sizeof(int));
mat[i * colunas + j] = valor;
```

- **Array de ponteiros** (não contíguo):

```
int **mat = malloc(linhas * sizeof *mat);
for (int i = 0; i < linhas; i++)
    mat[i] = malloc(colunas * sizeof **mat);
```

6. Percorrer arrays

```
for (int i = 0; i < n; i++)
    printf("%d ", arr[i]);
```

7. Manipulação comum

- **Rotacionar**: três reversões.
- **Reverter**: troca simétrica de elementos.
- **Remover elementos**: mover valores para frente e reduzir tamanho lógico.

Exercício A1 — `index_of` (enunciado + solução)

Enunciado. Implemente uma função em C que receba um array `a` de inteiros, seu tamanho `n` e um valor `x`, e retorne o **índice da primeira ocorrência** de `x`. Se não existir, retorne `-1`.

Assinatura sugerida

```
int index_of(const int *a, size_t n, int x);
```

Restrições

- Não modificar o array (use `const int *`).
- Complexidade de tempo O(n) e espaço extra O(1).
- Tratar `n=0` retornando `-1`.
- Evitar *undefined behavior* (checar ponteiro nulo se necessário).

Solução (sua versão final, com `main`)

```
#include <stdio.h>

int index_of(const int *arr, size_t n, int x);

int main() {
    int arr_1[3] = {3, 5, 7};
    int arr_2[3] = {1, 1, 1};
    int *arr_3 = NULL;

    printf("First array [3, 5, 7] -> x = 5: %d (SHOULD BE 1)
", index_of(arr_1, sizeof(arr_1) / sizeof(arr_1[0]), 5));
    printf("Second array [1, 1, 1] -> x = 2: %d (SHOULD BE -1)
", index_of(arr_2, sizeof(arr_2) / sizeof(arr_2[0]), 2));
    printf("Third array [] -> x = 9: %d (SHOULD BE -1)", index_of(arr_3, 0, 9));

    return 0;
}

int index_of(const int *arr, size_t n, int x) {
    int index = -1;
    for (size_t i = 0; i < n; i++) {
        if (*(arr + i) == x) {
            index = (int)i;
            break;
        }
    }
    return index;
}
```

Exercício A2 — Janela Deslizante (max_window_sum)

Intuição (Analogia)

Imagine uma **esteira de supermercado**. Cada produto tem um preço e sua **cesta** só comporta **k produtos contíguos**.
Você quer descobrir qual posição da esteira (qual bloco contíguo de tamanho k) dá o **maior total**.

- **Esteira** → array `a[]`
- **Cesta (tamanho k)** → janela fixa de tamanho `k`
- **Deslizar a cesta** → sai 1 item pela esquerda e entra 1 item pela direita

Teste de mesa (passo a passo)

Array `a = [1, 3, -2, 5, 3, -1]`, `k = 3`

1. Soma inicial dos índices `0..2`:

```
[ 1 ][ 3 ][-2 ]  5   3  -1
soma = 1 + 3 + (-2) = 2
max  = 2
```

2. Desliza 1 à direita (sai `1`, entra `5`):

```
  1 [ 3 ][-2 ][ 5 ]  3  -1
soma = 2 - 1 + 5 = 6
max  = max(2, 6) = 6
```

3. Desliza de novo (sai `3`, entra `3`):

```
  1   3 [-2 ][ 5 ][ 3 ] -1
soma = 6 - 3 + 3 = 6
max  = 6
```

4. Desliza de novo (sai `-2`, entra `-1`):

```
  1   3   -2 [ 5 ][ 3 ][-1]
soma = 6 - (-2) + (-1) = 7
max  = 7  ← melhor subarray é [5, 3, -1]
```

Por que é O(n) e não O(n·k)?

- Força bruta: recalcula a soma de cada janela do zero → O(n·k)
- Janela deslizante: **reaproveita a soma anterior**, atualizando só 2 itens (o que sai e o que entra) → **O(n)**

Enunciado & Restrições

Problema. Dado um array `a` de tamanho `n` e um inteiro `k` com `1 ≤ k ≤ n`, encontre a **maior soma** de um subarray **contíguo** de tamanho exato `k`.

Restrições / Regras:

- Não usar bibliotecas externas para resolver diretamente.
- **Tempo alvo:** O(n) — use a técnica de **janela deslizante**.
- **Espaço extra:** O(1).
- Lide com entradas válidas (`a != NULL`, `1 ≤ k ≤ n`); se quiser tratar erros, descreva o comportamento.
- Pense em casos de teste: `k=1`, `k=n`, valores negativos no array, todos iguais, mistura de positivos/negativos.

Solução (sua versão final, com `main`)

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

void get_max_window_sum(const int *arr, size_t size, size_t k);

int main() {
    int arr[] = {1, 3, -2, 5, 3, -1};

    // Should be 7
    get_max_window_sum(arr, sizeof(arr) / sizeof(arr[0]), 3);

    // Should be 5
    get_max_window_sum(arr, sizeof(arr) / sizeof(arr[0]), 1);

    // Should be -3
    int arr2[] = {-1, -1, -1, -1, -1, -1};
    get_max_window_sum(arr2, sizeof(arr2) / sizeof(arr2[0]), 3);

    // Should be nothing
    get_max_window_sum(NULL, 0, 3);

    return 0;
}

void get_max_window_sum(const int *arr, size_t size, size_t k) {
    if (arr == NULL || 1 > k || k > size) {
        printf("Invalid parameters! Null array or invalid subarray size.\n");
        return;
    }

    // good practice to avoid overflow in big sums
    long long sum = 0;

    // first we get the sum of the first window
    for (size_t i = 0; i < k; i++) {
        sum += arr[i];
    }
    printf("Initial sum: %lld\n", sum);
    long long best = sum;
    size_t best_start = 0;

    // now we apply the sliding window
    for (size_t i = k; i < size; i++) {
        sum = sum - arr[i - k] + arr[i];

        if (sum > best) {
            best = sum;
            best_start = i - k + 1;
        }
    }

    printf("Final maximum sum: %lld\n", best);
    printf("Maximum window sum: [");
    for (size_t i = 0; i < k; i++) {
        printf(
            "%d%s",
            arr[best_start + i],
            i + 1 < k ? ", " : ""
        );
    }
    printf("]\n");
    return;
}
```

Exercício A3 — Rotacionar array à direita (three reversals)

Revisão Teórica

Rotacionar um array à direita por `k` pode ser feito em O(n) tempo e O(1) espaço usando **três reversões**:

1. Inverta o array inteiro.
2. Inverta os `k` primeiros elementos.
3. Inverta os `n-k` elementos restantes. Trate `k %= n` e os casos triviais (`n == 0` ou `k == 0`).

Enunciado

Dado um array `a` de tamanho `n` e um inteiro `k`, rotacione `a` **à direita** por `k` posições **in-place**, usando a técnica das três reversões.

Resolução (código do aluno, sem modificações)

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

void print_arr(int *a, size_t size);

void reverse(int *a, size_t l, size_t r);

void rotate_right(int *a, size_t size, size_t k);

void test_rotation(int *a, size_t size, size_t k);

int main() {
  // basic cases
  int a1[] = {1, 2, 3, 4, 5};
  size_t size1 = sizeof(a1) / sizeof(a1[0]);
  test_rotation(a1, size1, 2); // Expected [4, 5, 1, 2, 3]

  int a2[] = {10, 20, 30, 40};
  size_t size2 = sizeof(a2) / sizeof(a2[0]);
  test_rotation(a2, size2, 1); // Expected [40, 10, 20, 30]

  int a3[] = {5, 6, 7};
  size_t size3 = sizeof(a3) / sizeof(a3[0]);
  test_rotation(a3, size3, 3); // Expected [5, 6, 7]

  // edge cases
  int a4[] = {1, 2, 3, 4, 5};
  size_t size4 = sizeof(a4) / sizeof(a4[0]);
  test_rotation(a4, size4, 7); // Expected [4, 5, 1, 2, 3]

  int a5[] = {42};
  size_t size5 = sizeof(a5) / sizeof(a5[0]);
  test_rotation(a5, size5, 5); // Expected [42]

  int a6[] = {};
  size_t size6 = sizeof(a6) / sizeof(a6[0]);
  test_rotation(a6, size6, 3); // Expected []

  // validators
  int a7[] = {1, 2, 3};
  size_t size7 = sizeof(a7) / sizeof(a7[0]);
  test_rotation(a7, size7, 0); // Expected [1, 2, 3]

  int a8[] = {9, 9, 9, 9};
  size_t size8 = sizeof(a8) / sizeof(a8[0]);
  test_rotation(a8, size8, 2); // Expected [9, 9, 9, 9]

  return 0;
}
```

```
void print_arr(int *a, size_t size) {
    printf("[");
    for (size_t i = 0; i < size; i++) {
        printf("%d%s", a[i], i + 1 < size ? ", " : "");
    }
    printf("]

");
}

void reverse(int *arr, size_t l, size_t r) {
  while (l < r) {
    int tmp = arr[r];
    arr[r] = arr[l];
    arr[l] = tmp;
    l++;
    r--;
  }
}

void rotate_right(int *arr, size_t size, size_t k) {
  // data normalization
  if (size == 0 || (k %= size) == 0) {
    print_arr(arr, size);
    return;
  }

  if (k > size) k %= size;

  // first we fully reverse the array - first reversion
  reverse(arr, 0, size - 1);

  // then we reverse only the first k elements - second reversion
  reverse(arr, 0, k - 1);

  // finally we reverse the last elements - third reversion
  reverse(arr, k, size - 1);

  print_arr(arr, size);
}

void test_rotation(int *arr, size_t size, size_t k) {
  rotate_right(arr, size, k);
  return;
}
```

Exercício A4 — Remover duplicatas em array ordenado (dois ponteiros)

Revisão Teórica

- **Pré-condição:** array em **ordem não decrescente**; duplicatas ficam **adjacentes**.
- **Dois ponteiros:** `slow` escreve a parte compactada; `fast` lê o resto. Início **`fast = slow + 1`** (se `n>0`).
- **Movimento:** se `a[fast] != a[slow]`, faça `slow++` e `a[slow] = a[fast]`.
- **Tamanho lógico:** no final, `slow + 1` (se `n>0`).
- **Complexidade:** tempo **O(n)**; espaço extra **O(1)**.
- **Armadilhas:** laço deve ser **`fast < n`** (evita acesso inválido); tratar `n==0`; não usar `const int *` (modifica array).

Analogia

Catálogo de camisetas já **ordenadas por tamanho**: percorra e **anote apenas o primeiro** de cada tamanho. O catálogo final ocupa as **primeiras posições**.

Teste de Mesa

`a = [1, 1, 2, 2, 2, 3, 4, 4]` → únicos nas **primeiras 4** posições: `[1,2,3,4]` (resto irrelevante).
Casos-limite: `[]→0`, `[7]→1`, `[1,2,3]→3`, `[5,5,5]→1`.

Enunciado & Restrições

Implemente `size_t dedup_sorted_inplace(int *a, size_t n);` que **remove duplicatas in-place** em array **ordenado** e retorna o **novo tamanho lógico**.
Restrições: **O(n)** tempo, **O(1)** espaço extra; tratar `n==0`; conteúdo após o tamanho lógico é irrelevante.

Resolução (código do aluno, sem modificações)

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

int dedup_sorted_inplace(int *arr, size_t n);

void print_arr(int *a, size_t size);

int main() {
  int a1[] = {1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 5};
  size_t size = sizeof(a1) / sizeof(a1[0]);

  printf("Initial array: ");
  print_arr(a1, size);

  printf("Deduped array: ");
  int logic_size = dedup_sorted_inplace(a1, size);

  if (logic_size == -1) return 0;

  print_arr(a1, logic_size);
  printf("Final logic size: %d", logic_size);
}

void print_arr(int *a, size_t size) {
  printf("[");
  for (size_t i = 0; i < size; i++) {
    printf(
      "%d%s",
      a[i],
      i + 1 < size ? ", " : ""
    );
  }
  printf("]\n\n");
}

int dedup_sorted_inplace(int *arr, size_t n) {
  if (n == 0) {
      printf("Invalid array");
      return -1;
  }

  int slow = 0;
  size_t fast = 1;

  while(fast < n) {
    if (arr[fast] != arr[slow]) {
      slow++;
      arr[slow] = arr[fast];
    }
    fast++;
  }

  return slow + 1;
}
```

Exercício A5 — Matriz 2D contígua (`malloc` + indexação linear)

Revisão Teórica

- **Objetivo:** representar uma matriz `rows x cols` em **um único bloco contíguo** de memória usando `malloc`.
- **Alocação contígua:** `int *buf = malloc(rows * cols * sizeof *buf);`
  - **Vantagens:** um único `malloc/free`, melhor **localidade de cache**, sem fragmentação entre linhas.
  - **Cuidados:** validar `buf != NULL`; atenção a overflow em `rows * cols` se usar tipos pequenos.
- **Layout row-major (C):** os elementos de uma linha ficam **lado a lado** em memória.
  - **Indexação linear:** `buf[i * cols + j]` acessa a célula `(i, j)`.
  - **Endereço:** `&buf[i * cols + j]` → compare com "`&buf[0] + (i*cols + j)`".
- **Comparação com `int**` (ponteiro para ponteiro):**
  - `int **a` **não garante contiguidade**: cada linha pode ser um bloco separado (vários `malloc`).
  - Em contígua, você tem **um** bloco; em `int**`, você tem **N+1** blocos (vetor de linhas + N linhas).
- **Passagem para funções:** passe o **ponteiro base**, mais `rows` e `cols`.
  - Ex.: `int get(const int *buf, size_t rows, size_t cols, size_t i, size_t j)`.
- **Inicialização:** `calloc` zera; `malloc` não zera (conteúdo indeterminado).
- **Liberação:** `free(buf);` (apenas **uma** vez).
- **Armadilhas comuns:**
  - **Off-by-one:** lembrar que `0 ≤ i < rows` e `0 ≤ j < cols`.
  - **Misturar `int**` com bloco contíguo:** `a[i][j]` só funciona se você modelar como `int (*a)[cols]` (VLA) ou um `int**` adequadamente inicializado; com `int *buf`, use **indexação linear**.
  - **Overflow:** se `rows*cols` pode extrapolar `size_t`, valide antes de multiplicar.

Analogia

Imagine uma **planilha** onde você "desenrola" todas as linhas em uma **fita contínua**. A célula `(i, j)` vira a posição **`i*cols + j`** nessa fita. Você caminha pela fita somando deslocamentos fixos de `cols` para pular de uma linha à próxima.

Teste de Mesa

`rows=3`, `cols=4` → índices válidos de `(0..2, 0..3)`
Preencha com `valor = i*cols + j`:

- Linha 0: `[0, 1, 2, 3]`
- Linha 1: `[4, 5, 6, 7]`
- Linha 2: `[8, 9, 10, 11]` Verificação: `buf[2*4 + 1] = buf[9] = 9` (célula `(2,1)`).

Enunciado & Restrições

Implemente uma API mínima para **matriz 2D contígua**:

1. `int* mat_alloc(size_t rows, size_t cols)` → aloca e retorna o buffer contíguo (ou `NULL`).
2. `void mat_free(int *buf)` → libera o buffer.
3. `int mat_get(const int *buf, size_t rows, size_t cols, size_t i, size_t j)` → lê `(i,j)` validando limites (defina política de erro).
4. `void mat_set(int *buf, size_t rows, size_t cols, size_t i, size_t j, int v)` → escreve `(i,j)` validando limites.
5. (Opcional) `void mat_fill_seq(int *buf, size_t rows, size_t cols)` → preenche `buf[k]=k` para facilitar depuração.

Restrições:

- Usar **um único bloco contíguo** (`malloc` / `calloc`) e **indexação linear** (`i*cols + j`).
- Tempo por acesso **O(1)**; espaço extra além do buffer deve ser **O(1)**.
- Tratar casos-limite (`rows==0` ou `cols==0` → retornar `NULL` na alocação, por exemplo).

Resolução (código do aluno, sem modificações)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdint.h>
#include <limits.h>

int* mat_alloc(size_t rows, size_t cols);

void mat_free(int *buf);

int mat_get(const int *buf, size_t rows, size_t cols, size_t row, size_t col);

void mat_set(int *buf, size_t rows, size_t cols, size_t row, size_t col, int v);

void fill_mat_seq(int *buf, size_t rows, size_t cols);

void print_mat(int *buf, size_t rows, size_t cols);

int main() {
```

```
int main() {
  int *buf = mat_alloc(2, 3);
  fill_mat_seq(buf, 2, 3);
  print_mat(buf, 2, 3);

  printf("
[MAT_GET] Row 2 | Col 3 | Return: %d
", mat_get(buf, 2, 3, 1, 2)); // Should be 6

  mat_set(buf, 2, 3, 1, 2, 10);

  printf("[MAT_GET] (After MAT_SET) Row 2 | Col 3 | Return: %d

", mat_get(buf, 2, 3, 1, 2)); // Should be 10

  print_mat(buf, 2, 3);

  mat_free(buf);
}

int* mat_alloc(size_t rows, size_t cols) {
  if (rows == 0 || cols == 0) return NULL;

  if (rows > SIZE_MAX / cols) return NULL;

  int *buf = calloc(rows * cols, sizeof *buf);

  if (!buf) return NULL;

  return buf;
}

void mat_free(int *buf) {
  if (buf != NULL) free(buf);
}

int mat_get(const int *buf, size_t rows, size_t cols, size_t row, size_t col) {
  if (row >= rows || col >= cols || buf == NULL) return INT_MIN;

  return buf[row * cols + col];
}

void mat_set(int *buf, size_t rows, size_t cols, size_t row, size_t col, int v) {
  if (row >= rows || col >= cols || buf == NULL) return;

  buf[row * cols + col] = v;
}

void fill_mat_seq(int *buf, size_t rows, size_t cols) {
  if (!buf) return;

  int v = 1;
  for (size_t i = 0; i < rows; i++) {
    for (size_t j = 0; j < cols; j++) {
      buf[i * cols + j] = v;
      v++;
    }
  }
}

void print_mat(int *buf, size_t rows, size_t cols) {
  if (!buf) return;

  for (size_t i = 0; i < rows; i++) {
    printf("%zuº row: [", i + 1);
    for (size_t j = 0; j < cols; j++) {
```

```
      printf("%d%s", buf[i * cols + j], j + 1 < cols ? ", " : "");
    }
    printf("]
");
  }
}
```