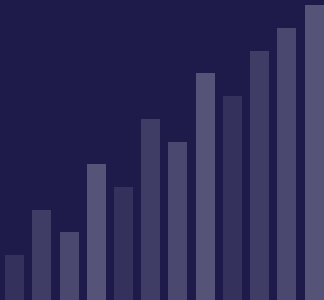

BASE 2 · ANEXO: BASE PRÁTICA EM C

Ponteiros: endereços, aritmética e ponteiro para ponteiro

Kernighan e Ritchie, *C: a linguagem de programação*,
capítulo 5

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Usar & e * e entender o que um ponteiro guarda
- ▶ Relacionar vetores e ponteiros pela aritmética de ponteiros
- ▶ Passar parâmetros por referência e alterar um ponteiro dentro de uma função (int **)
- ▶ Alocar matrizes dinâmicas e medir o efeito da ordem de percurso



Teoria

Endereços, aritmética, vetores, referência e ponteiro duplo



Prática em C

Troca por referência, matriz dinâmica e vetor alocado via int **



Experimento

Percorrer uma matriz por linhas ou por colunas



Conexões

Sistemas operacionais, compiladores, hardware e bancos de dados

Endereços e indireção

- ▶ Toda variável mora num **endereço**. `&x` é o endereço de `x`; um ponteiro é uma variável que guarda um endereço.
- ▶ `int *p = &x;` então `*p` **é** `x`: ler e escrever `*p` lê e escreve `x`.
- ▶ `NULL`: "não aponta para nada". Desreferenciar `NULL` derruba o programa (*segmentation fault*).
- ▶ C passa tudo **por valor**: para uma função alterar `x`, ela recebe `&x` (`void troca(int *a, int *b)`).

No Cormen, "`x.prox` é um ponteiro; " `extscnil` é `exttttNULL`. Toda estrutura ligada do curso é feita disto.

Vetores e aritmética de ponteiros

- ▶ O nome de um vetor é o endereço do primeiro elemento: $v == \&v[0]$.
- ▶ $p + i$ anda i **elementos** (não bytes): $*(p + i)$ é $p[i]$.
- ▶ $p2 - p1$ dá quantos elementos há entre dois ponteiros do mesmo vetor.
- ▶ Ao passar um vetor para uma função, passa-se só o endereço: a função não sabe o tamanho; passe n junto.

Expressão	Tipo	Significado
$v[i], *(v+i)$	<code>int</code>	o elemento i
$\&v[i], v+i$	<code>int *</code>	o endereço dele
$m[i][j]$	<code>int</code>	com <code>int **m</code> : $*(*(m+i)+j)$

Ponteiro para ponteiro

- ▶ **Alterar um ponteiro do chamador:** se `aloca(int *p)` faz `p = malloc(...)`, o chamador nunca vê. É preciso `void aloca(int **p)` e `*p = malloc(...)`, chamada como `aloca(&v)`.
- ▶ **Matriz dinâmica:** `int **m` é um vetor de ponteiros para linhas. Alocar as linhas num bloco contíguo (`m[i] = m[0] + i*col`) torna a matriz amiga do cache.
- ▶ **Inserir no início de uma lista sem cabeçalho:** `insere(No **cabeca, int k)` pode mudar a cabeça.
- ▶ Em C, uma matriz é guardada **linha por linha** (*row-major*): `m[i][j]` e `m[i][j+1]` são vizinhos na memória; `m[i+1][j]`, não.



Prática em C

- ▶ troca por referência
- ▶ Matriz dinâmica com um bloco contíguo
- ▶ `aloca_vetor(int **p, n)` altera o ponteiro do chamador

Ponteiros em C (1/3)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // passagem por referência: recebe endereços
5  void troca(int *a, int *b) {
6      int t = *a;                                // *a: o valor guardado no endereço a
7      *a = *b;
8      *b = t;
9  }
10
11  /* Para que uma função ALTERE um ponteiro do chamador, ela recebe o endereço do
12     ponteiro: um ponteiro para ponteiro (int **). */
13  void aloca_vetor(int **p, int n) {
14      *p = malloc(n * sizeof **p);
15      for (int i = 0; i < n; i++) (*p)[i] = 10 * i;
16  }
```

Ponteiros em C (2/3)

```
17
18 /* Matriz dinâmica: um vetor de ponteiros para linhas (int **). Uma alocação para
19 os ponteiros e outra, contígua, para todos os elementos. */
20 int **cria_matriz(int lin, int col) {
21     int **m = malloc(lin * sizeof *m);
22     m[0] = malloc((size_t)lin * col * sizeof **m);
23     for (int i = 1; i < lin; i++) m[i] = m[0] + i * col;    // aritmética de ponteiros
24     return m;
25 }
26
27 void libera_matriz(int **m) { free(m[0]); free(m); }
28
29 int main(void) {
30     int a, b;
31     if (scanf("%d %d", &a, &b) != 2 || a < 1 || b < 1) return 1;
32     int lin = a, col = b;
```

Ponteiros em C (3/3)

```
33     troca(&a, &b);                                // &a: o endereço de a
34     printf("troca: a = %d, b = %d\n", a, b);
35     int v[3] = {7, 8, 9}, *q = v;                 // o nome do vetor é o endereço do 1º
36     printf("v[1] = %d = *(v + 1) = %d = q[1] = %d\n", v[1], *(v + 1), q[1]);
37     int **m = cria_matriz(lin, col);
38     for (int i = 0; i < lin; i++)
39         for (int j = 0; j < col; j++) m[i][j] = 10 * i + j;
40     printf("matriz %d x %d: m[%d][%d] = %d\n", lin, col, lin - 1, col - 1,
41           m[lin - 1][col - 1]);
42     libera_matriz(m);
43     int *w = NULL;
44     aloca_vetor(&w, 3);                            // w muda dentro da função
45     printf("vetor via ponteiro duplo: %d %d %d\n", w[0], w[1], w[2]);
46     free(w);
47     return 0;
48 }
```



Experimento

- ▶ Matriz $n \times n$ de inteiros, n de 256 a 4.096
- ▶ Mesma soma, duas ordens de percurso
- ▶ Tempo por elemento, melhor de 3

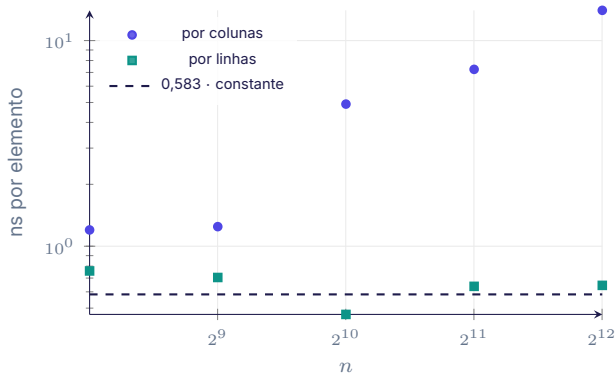
Linhas ou colunas?

Hipótese. No modelo RAM, as duas ordens fazem as mesmas n^2 somas: o mesmo tempo. Na máquina real, a matriz está guardada por linhas: percorrer por linhas lê endereços vizinhos (aproveita cada linha de cache de 64 bytes); por colunas, salta $4n$ bytes a cada acesso. Esperamos linhas com custo **constante** e colunas cada vez piores.

Método

1. Matriz em bloco contíguo ($m[i] = m[0] + i*n$)
2. $n = 256, 512, \dots, 4096$ (até 64 MB)
3. Mínimo de 3 repetições
4. Tempo dividido por n^2

Tempo por elemento



n	medido	razão
256,0	0,758	0,758
512,0	0,705	0,705
1.024	0,466	0,466
2.048	0,638	0,638
4.096	0,645	0,645
4.096	0,645	0,645

razão = medido $\div$ constante

✓ desvio máx. 20,1%

Por linhas: cerca de 0,3 ns, constante.
Por colunas: de 1 ns a 10 ns quando a matriz deixa de caber no cache.

EXPERIMENTO

O que o experimento mostrou

A mesma conta, na ordem errada, fica **30 vezes** mais lenta para $n = 4096$. Ponteiros dão acesso direto à memória, e a memória tem geografia: vizinhos são baratos, saltos são caros. Nas próximas aulas, a mesma lição aparece em listas ligadas (aula 14) e em grafos (aula 29).

➤ Reproduza: `gcc -O2 base2_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Sistemas operacionais
- ▶ Compiladores
- ▶ Hardware e GPUs
- ▶ Bancos de dados

Onde isso aparece na computação

Sistemas operacionais

O kernel manipula páginas, processos e dispositivos por ponteiros; um ponteiro inválido no kernel derruba a máquina inteira.

Compiladores

Otimizadores trocam a ordem de laços (*loop interchange*) para percorrer matrizes por linhas; a análise de *aliasing* decide se dois ponteiros podem se sobrepor.

Hardware e GPUs

Bibliotecas de álgebra linear (BLAS) e GPUs dividem matrizes em blocos que cabem no cache: o mesmo efeito do experimento, explorado ao máximo.

Bancos de dados

Bancos orientados a linhas (OLTP) e a colunas (OLAP, como DuckDB) são as duas ordens do experimento: cada um vence num tipo de consulta.

Para casa

Exercícios

1. Escreva `void minmax(const int *v, int n, int *mn, int *mx)` que devolve mínimo e máximo pelos ponteiros.
2. Implemente `strlen` só com aritmética de ponteiros (sem índices).
3. Escreva `int **le_matriz(int *lin, int *col)` que lê as dimensões e os valores e devolve a matriz.
4. Implemente a inserção na cabeça de uma lista com `void insere(No **cabeca, int k)`.
5. Transponha uma matriz $n \times n$ "no lugar e meça o tempo: o acesso por colunas aparece de novo?

Leitura

Kernighan e Ritchie, *C: a linguagem de programação*, capítulo 5 (ponteiros e vetores).

Código

josedihego.net/estruturas