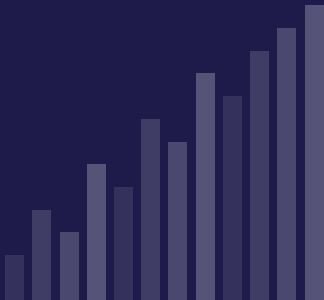


BASE 4 · ANEXO: BASE PRÁTICA EM C

Funções: recursão, ponteiros para funções e código genérico

Kernighan e Ritchie, *C: a linguagem de programação*,
seções 4.10, 5.11 e B.5



Roteiro da aula

Ao final, você será capaz de

- ▶ Entender a pilha de chamadas e o custo de memória da recursão
- ▶ Declarar e usar ponteiros para funções
- ▶ Escrever código genérico com `void *` e `qsort`
- ▶ Medir os bytes de pilha por chamada e as comparações do `qsort`

Teoria

Quadros de pilha, ponteiros para funções, `void *` e `qsort`

Prática em C

Comparadores, tabela de funções, troca genérica e fatorial

Experimento

Pilha por chamada recursiva; chamadas do comparador

Conexões

Sistemas operacionais, interfaces gráficas, bibliotecas de IA

A pilha de chamadas

- ▶ Cada chamada empilha um **quadro** (*stack frame*): parâmetros, variáveis locais, endereço de retorno.
- ▶ Na recursão, há um quadro **por nível**: `fatorial(10000)` tem 10.000 quadros vivos ao mesmo tempo.
- ▶ A pilha é pequena (8 MB no Linux, por padrão): recursão de profundidade n com n de milhões causa *stack overflow*.
- ▶ Recursão de profundidade $O(\lg n)$ (merge sort, busca binária, árvores balanceadas) é segura; de profundidade $O(n)$ (quicksort no pior caso, DFS) pede cuidado.

Por isso o quicksort do complemento 1 recorre no lado menor, e a DFS da aula 29 usa uma pilha explícita.

Ponteiros para funções

- ▶ O nome de uma função é o endereço do seu código: pode ir para uma variável.
- ▶ `int (*cmp)(const void *, const void *) = crescente;    depois, cmp(&a, &b).`
- ▶ Com `typedef int (*Comparador)(const void *, const void *)`; a declaração fica legível.
- ▶ Usos: **callbacks** (o `qsort` chama a sua função de comparação), **tabelas de despacho** (`op[k](x, y)`), estratégias trocáveis.

É assim que se escreve “uma ordenação que ordena qualquer coisa: o algoritmo é fixo, a comparação é um parâmetro.

Código genérico com `void *`

- ▶ `void *` é “ponteiro para qualquer coisa”; não pode ser desreferenciado sem conversão.
- ▶ `qsort(base, n, tamanho, cmp)` ordena n elementos de tamanho bytes: não sabe o tipo, só chama `cmp`.
- ▶ Dentro do comparador: `int x = *(const int *)a;`
- ▶ `memcpy(destino, origem, bytes)` copia qualquer tipo: com ela se escreve uma troca genérica.
- ▶ Cuidado: o compilador já não confere os tipos. Um comparador de `double` usado num vetor de `int` compila e dá lixo.



Prática em C

- ▶ Comparadores crescente e decrescente para o qsort
- ▶ Tabela de ponteiros para funções
- ▶ Troca genérica com memcpy e fatorial recursivo

Funções e ponteiros para funções (1/4)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 /* Comparadores no formato que o qsort espera: recebem void * (qualquer tipo)
6    e devolvem negativo, zero ou positivo. */
7 int crescente(const void *a, const void *b) {
8     int x = *(const int *)a, y = *(const int *)b;
9     return (x > y) - (x < y);           // evita o estouro de x - y
10 }
11 int decrescente(const void *a, const void *b) { return crescente(b, a); }
```

Funções e ponteiros para funções (2/4)

```
13  /* Ponteiro para função: uma variável que guarda "qual função chamar". */
14  typedef int (*Comparador)(const void *, const void *);
15
16  /* Troca genérica: funciona para qualquer tipo, dado o tamanho em bytes. */
17  void troca(void *a, void *b, size_t tam) {
18      unsigned char tmp[64];
19      memcpy(tmp, a, tam);
20      memcpy(a, b, tam);
21      memcpy(b, tmp, tam);
22  }
23
24  // recursão: cada chamada tem seu quadro na pilha
25  long fatorial(int n) {
26      return n <= 1 ? 1 : n * fatorial(n - 1);
27  }
```

Funções e ponteiros para funções (3/4)

```
28
29 void imprime(const char *nome, const int v[], int n) {
30     printf("%s:", nome);
31     for (int i = 0; i < n; i++) printf(" %d", v[i]);
32     printf("\n");
33 }
34
35 int main(void) {
36     int n;
37     if (scanf("%d", &n) != 1 || n < 1 || n > 20) return 1;
38     int v[20];
39     for (int i = 0; i < n; i++)
```

Funções e ponteiros para funções (4/4)

```
40     if (scanf("%d", &v[i]) != 1) return 1;
41     Comparador ordens[2] = {crescente, decrescente};           // tabela de funções
42     const char *nomes[2] = {"crescente", "decrescente"};
43     for (int k = 0; k < 2; k++) {
44         qsort(v, n, sizeof v[0], ordens[k]);                  // qsort chama de volta
45         imprime(nomes[k], v, n);
46     }
47     double x = 1.5, y = 2.5;
48     troca(&x, &y, sizeof x);
49     printf("troca genérica: x = %.1f, y = %.1f\n", x, y);
50     printf("fatorial de %d = %ld\n", n, fatorial(n));
51     return 0;
52 }
```



Experimento

- ▶ Recursão de profundidade n : endereço da local mais funda
- ▶ Contador dentro do comparador do qsort
- ▶ n até 16 mil (pilha) e até um milhão (qsort)

Recursão e callbacks, medidos

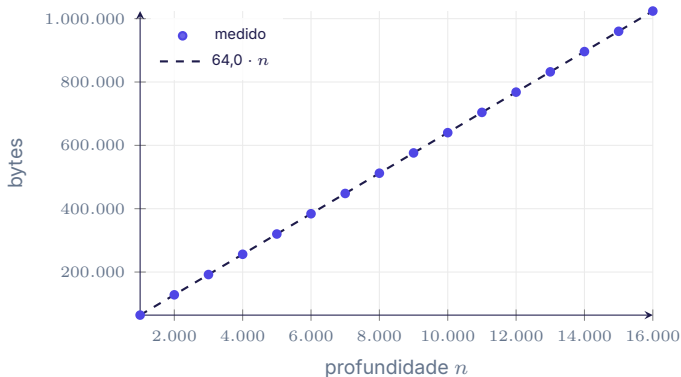
Hipótese. Cada nível da recursão ocupa um quadro de tamanho fixo: a pilha usada deve ser **exatamente linear** em n . O qsort do glibc é, na verdade, um merge sort: deve chamar o comparador $\approx n \lg n - 1,26 n$ vezes, como o merge sort da aula 4.

Método

1. Função recursiva com 40 bytes de locais (noinline)
2. Pilha = endereço inicial — endereço mais fundo
3. qsort de n inteiros aleatórios
4. Contagem exata

EXPERIMENTO

Bytes de pilha na recursão



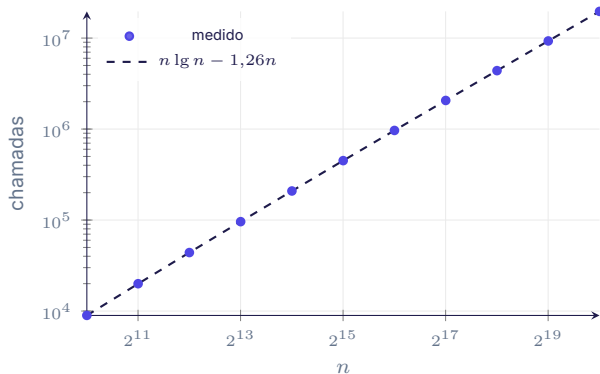
n	medido	razão
1.000	64.071	64,1
4.000	256.071	64,0
7.000	448.071	64,0
10.000	640.071	64,0
13.000	832.071	64,0
16.000	1.024.071	64,0

razão = medido $\div$ n

✓ desvio máx. 0,0%

Uma reta perfeita: o coeficiente é o tamanho do quadro (64 bytes neste compilador). Com 8 MB de pilha, cabem cerca de 130 mil níveis.

Chamadas do comparador



n	medido	razão
1.024	8.961	1,002
4.096	43.936	0,999
16.384	208.691	1,000
65.536	965.706	1,000
262.144	4.386.694	1,000
1.048.576	19.646.016	1,000

razão = medido $\div$ $n \lg n - 1,26n$

✓ desvio máx. 0,0%

Exatamente a fórmula do merge sort da aula 4: o qsort do glibc intercala.

EXPERIMENTO

O que o experimento mostrou

A pilha cresce exatamente em linha reta com a profundidade (um quadro de tamanho fixo por chamada), e o qsort chama o comparador $n \lg n - 1,26 n$ vezes: a fórmula do merge sort. Medir revela o que a biblioteca faz por dentro: o nome diz “quick”, mas o comportamento é de merge sort.

> Reproduza: `gcc -O2 base4_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Sistemas operacionais
- ▶ Interfaces gráficas
- ▶ Programação orientada a objetos
- ▶ Inteligência artificial

Onde isso aparece na computação

Sistemas operacionais

Tratadores de sinais e de interrupções, `pthread_create` e os drivers de dispositivo (`struct file_operations`) são tabelas de ponteiros para funções.

Interfaces gráficas

Botões e eventos chamam funções registradas (*callbacks*): GTK, o DOM do navegador e o Android funcionam assim.

Programação orientada a objetos

Métodos virtuais do C++ são ponteiros para funções numa tabela (*vtable*); o mesmo padrão aparece no kernel escrito em C.

Inteligência artificial

Funções de ativação, de perda e otimizadores são passados como parâmetros: um treinador genérico recebe "qual função usar".

Para casa

Exercícios

1. Escreva um comparador que ordene Aluno por nota decrescente e, no empate, por nome.
2. Implemente `void aplica(int *v, int n, int (*f)(int))` que substitui cada elemento por `f(v[i])`.
3. Faça uma calculadora com uma tabela `double (*op[4])(double, double)` indexada pelo operador.
4. Escreva `void *busca_linear(void *base, size_t n, size_t tam, const void *chave, Comparador cmp)`.
5. Descubra, aumentando n , a profundidade em que a recursão do experimento estoura a pilha. Bate com 8 MB / 64 bytes?

Leitura

Kernighan e Ritchie, *C: a linguagem de programação*, seções 4.10 (recursão), 5.11 (ponteiros para funções) e B.5 (qsort).

Código

josedihego.net/estruturas