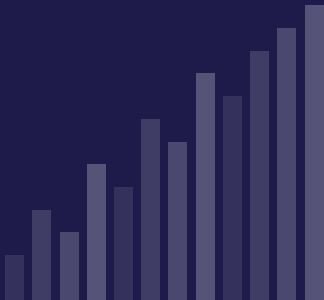

BASE 1 · ANEXO: BASE PRÁTICA EM C

Memória: pilha, heap e alocação dinâmica

Kernighan e Ritchie, *C: a linguagem de programação*,
seções 5.4 e 8.7

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Distinguir a pilha (variáveis locais) do heap (alocação dinâmica)
- ▶ Usar malloc, calloc, realloc e free corretamente
- ▶ Reconhecer vazamentos, ponteiros soltos e liberações duplas
- ▶ Medir quanto um malloc pequeno custa em bytes e em tempo



Teoria

O mapa da memória de um processo e as quatro funções de alocação



Prática em C

Vetor no heap, calloc e crescimento com realloc



Experimento

Bytes e nanossegundos por malloc de 16 bytes

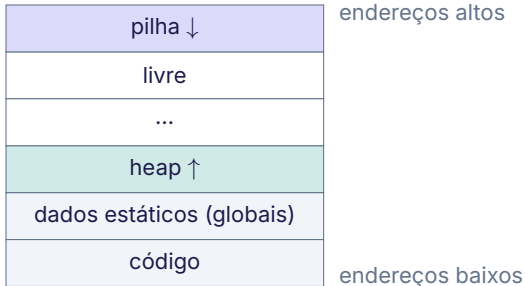


Conexões

Sistemas operacionais, segurança, linguagens e sistemas embarcados

O mapa da memória de um processo

- ▶ **Código e dados estáticos:** o programa e as variáveis globais.
- ▶ **Heap:** cresce para cima; é de onde vem o malloc. A memória fica até o free.
- ▶ **Pilha:** cresce para baixo; guarda variáveis locais e endereços de retorno. Some quando a função retorna.



Estruturas de dados que crescem (listas, árvores, tabelas) vivem no heap: o tamanho só é conhecido durante a execução.

As quatro funções

Chamada	O que faz
<code>p = malloc(n * sizeof *p)</code>	reserva n elementos, conteúdo indefinido; NULL se falhar
<code>p = calloc(n, sizeof *p)</code>	reserva e zera
<code>q = realloc(p, m * sizeof *p)</code>	muda o tamanho, preservando o conteúdo (pode mudar o endereço)
<code>free(p)</code>	devolve a memória; p continua apontando para lá ("solto")

- ▶ `sizeof *p` em vez de `sizeof(int)`: continua certo se o tipo de p mudar.
- ▶ Nunca `p = realloc(p, ...)`: se falhar, o NULL sobrescreve o único ponteiro para o bloco antigo.

Os erros clássicos

- ▶ **Vazamento:** perder o último ponteiro para um bloco sem chamar free. O programa cresce até acabar a memória.
- ▶ **Ponteiro solto** (*dangling*): usar um bloco depois do free, ou o endereço de uma variável local depois que a função retornou.
- ▶ **Liberação dupla:** free duas vezes no mesmo bloco corrompe o alocador.
- ▶ **Estouro:** escrever além do tamanho pedido ($v[n]$ num vetor de n).

Ferramentas: valgrind ./prog aponta vazamentos e acessos inválidos; gcc -fsanitize=address faz o mesmo ao executar.

Cerca de 70% das falhas de segurança graves em produtos da Microsoft e do Chrome vêm desses erros de memória.



Prática em C

- ▶ Por que não devolver o endereço de uma local
- ▶ Vetor no heap, calloc que zera
- ▶ realloc seguro com ponteiro temporário

Alocação dinâmica em C (1/4)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Variáveis locais ficam na PILHA e somem quando a função retorna; o que vem de
5     malloc fica no HEAP até alguém chamar free. Devolver o endereço de uma local
6     é um erro clássico: a memória já não é sua. */
7  int *vetor_na_pilha_errado(void) {
8     int v[4] = {1, 2, 3, 4};
9     (void)v;                                     // não faça: return v;
10    return NULL;
11 }
```

Alocação dinâmica em C (2/4)

```
13 int *vetor_no_heap(int n) { // certo: quem chama dá free
14     int *v = malloc(n * sizeof *v); // sizeof *v: segue o tipo de v
15     if (!v) { perror("malloc"); exit(1); } // malloc pode falhar
16     for (int i = 0; i < n; i++) v[i] = i + 1;
17     return v;
18 }
19
20 int main(void) {
21     int n;
22     if (scanf("%d", &n) != 1 || n < 1) return 1;
23     vetor_na_pilha_errado();
24     int *v = vetor_no_heap(n);
```

Alocação dinâmica em C (3/4)

```
25  long soma = 0;
26  for (int i = 0; i < n; i++) soma += v[i];
27  printf("soma = %ld\n", soma);
28  int *zeros = calloc(n, sizeof *zeros);      // calloc: aloca e zera
29  printf("calloc zero: zeros[0] = %d\n", zeros[0]);
30  // realloc: cresce preservando o conteúdo
31  int cap = 1, tam = 0;
32  int *w = malloc(cap * sizeof *w);
33  for (int i = 0; i < n; i++) {
34      if (tam == cap) {
35          int *novo = realloc(w, 2 * cap * sizeof *w);
36          if (!novo) { free(w); return 1; }    // nunca w = realloc(w, ...) direto
37          w = novo;
```

Alocação dinâmica em C (4/4)

```
38         cap *= 2;
39     }
40     w[tam++] = v[i];
41 }
42 printf("capacidade final %d\n", cap);
43 free(v); // cada malloc/calloc/realloc: um free
44 free(zeros);
45 free(w);
46 v = NULL; // evita usar um ponteiro solto
47 printf("tudo liberado\n");
48 return 0;
49 }
```



Experimento

- ▶ n blocos de 16 bytes, n de 4 mil a 4 milhões
- ▶ Bytes em uso medidos com `mallinfo2`
- ▶ Tempo por `malloc`

O preço de um malloc pequeno

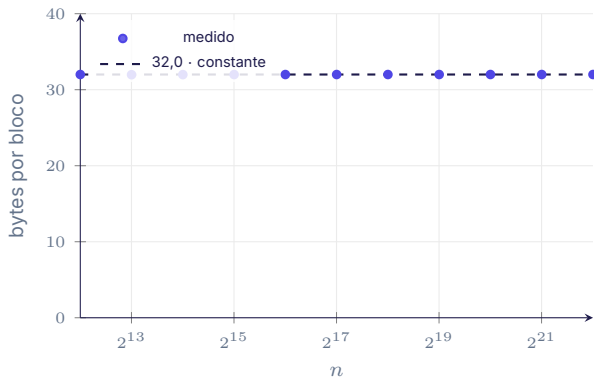
Hipótese. Cada bloco do alocador do glibc carrega um **cabeçalho** (8 bytes) e é alinhado a 16 bytes: um pedido de 16 bytes deve ocupar 32. O custo deve ser **constante** por bloco, em bytes e em tempo: alocar um milhão de nós de lista custa o dobro da memória dos dados.

Método

1. `mallinfo2().uordblks`: bytes do heap em uso, antes e depois
2. $n = 2^{12}, \dots, 2^{22}$ blocos
3. Tempo total das n chamadas dividido por n
4. Todos os blocos liberados a cada rodada

🧪 EXPERIMENTO

Bytes por bloco de 16 bytes



n	medido	razão
4.096	32,0	32,0
16.384	32,0	32,0
65.536	32,0	32,0
262.144	32,0	32,0
1.048.576	32,0	32,0
4.194.304	32,0	32,0

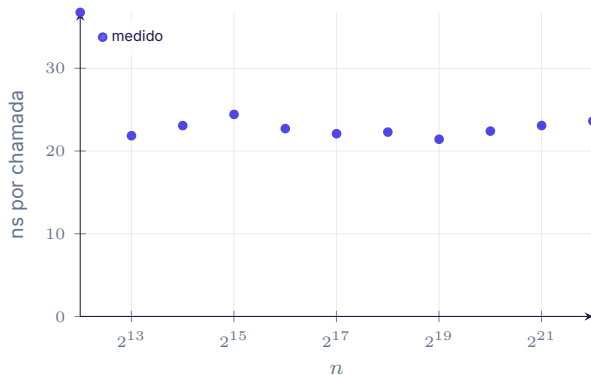
razão = medido $\div$ constante

✓ desvio máx. 0,0%

Exatamente 32 bytes: 16 de dados, 8 de cabeçalho e 8 de alinhamento.

🧪 EXPERIMENTO

Tempo por malloc



Dezenas de nanossegundos: 50 a 100 vezes o custo de uma soma. Alocar um nó por elemento tem preço.

EXPERIMENTO

O que o experimento mostrou

Cada malloc de 16 bytes ocupa **32 bytes** e custa dezenas de nanossegundos. Uma lista ligada de inteiros gasta muito mais memória e tempo que um vetor com os mesmos dados. É por isso que estruturas de alto desempenho alocam em blocos (*pools*) e por isso a aula 16 mostra listas guardadas em vetores.

➤ Reproduza: `gcc -O2 base1_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Sistemas operacionais
- ▶ Segurança
- ▶ Linguagens com coleta de lixo
- ▶ Sistemas embarcados

Onde isso aparece na computação

Sistemas operacionais

malloc pede páginas ao kernel com brk e mmap; a memória virtual e a paginação ficam por trás de cada ponteiro.

Segurança

Estouros de buffer e uso após free são as falhas mais exploradas da história; linguagens como Rust impedem esses erros já na compilação.

Linguagens com coleta de lixo

Java, Python e Go liberam a memória sozinhas (coletor de lixo); C dá controle total, e a responsabilidade junto.

Sistemas embarcados

Em microcontroladores com poucos KB, o heap fragmenta: por isso o padrão MISRA C proíbe malloc depois da inicialização.

Para casa

Exercícios

1. Escreva `int *copia(const int *v, int n)` que devolve uma cópia de `v` no heap.
2. Rode `base1_memoria.c` com `valgrind` e depois remova um `free`. O que ele relata?
3. Leia números até o fim da entrada num vetor que cresce com `realloc` (dobrando).
4. Repita o experimento com blocos de 8, 24, 32 e 100 bytes e tabele os bytes realmente usados.
5. Compile com `-fsanitize=address` um programa que acessa `v[n]` e leia a mensagem.

Leitura

Kernighan e Ritchie, C: a linguagem de programação, seções 5.4 e 8.7 (alocador); Cormen et al., seção 10.3.

Código

josedihego.net/estruturas