

Estrutura de Dados

Estrutura de Dados

Memória Dinamicamente Alocada

Sumário

1 Definição

2 Estrutura

3 Vetor

4 Matriz

5 Realocar

Definição

Memória dinamicamente alocada

Criar e manter estruturas dinâmicas de dados exige alocação dinâmica de memória: a capacidade de um programa obter, em tempo de execução, mais espaço de memória para conter novos nós e liberar espaço não mais necessário.

(Deitel; Deitel, 2011)

- › **sizeof(T)**: retorna o número de bytes que o tipo T ocupa em memória.
- › **malloc(n)**: aloca n bytes e retorna um void* para o bloco (ou NULL em falha).
- › **realloc(ptr, n)**: redimensiona o bloco apontado por ptr para n bytes (pode mover o bloco).
- › **free(ptr)**: libera a memória previamente alocada em ptr.
- › Todas estas funções estão em <stdlib.h>.

```
1
2 #include <stdlib.h>
3 typedef struct{
4     int a;
5     float b;
6     char c;
7 }Node;
8 int main(int argc, char const *argv[])
9 {
10     void* ptrNode = malloc(sizeof(Node));
11     return 0;
12 }
```

Código 1: Exemplo de uso da função malloc().

› Sem tipo:

- ›› Em funções: indica que não há valor de retorno.
- ›› Em declarações de parâmetro: funções sem parâmetros (C90) usam void para indicar lista de parâmetros vazia.

› Ponteiro genérico (void*):

- ›› Aponta para um bloco de memória sem tipo definido.
- ›› Pode ser convertido (cast) para qualquer outro ponteiro de objeto.

› void como tipo de retorno:

- ›› `void fun(void);` → função sem retorno e sem parâmetros (C90).
- ›› Até o ANSI C89, omitir o parâmetro (`fun()`) significava “lista de parâmetros desconhecida”.

› void* genérico:

- ›› Conversão implícita de qualquer `T*` para `void*`.
- ›› Para converter de volta, use cast explícito: `(T*)ptr`.
- ›› Não é permitida aritmética de ponteiros em `void*` (tamanho indefinido).

› Restrições adicionais:

- ›› `sizeof(void)` é inválido.
- ›› Não se pode declarar variáveis de tipo `void` nem retornar objetos `void`.

Estrutura

```
1
2 #include <stdlib.h>
3 typedef struct{
4     int a;
5     float b;
6     char c;
7 }Node;
8 int main(int argc, char const *argv[])
9 {
10     Node *ptrNode = malloc(sizeof(Node));
11     if(ptrNode != NULL){
12         ptrNode->a = 1;
13         ptrNode->b = 1.5;
14         ptrNode->c = '1';
15     }
16     free(ptrNode);
17     return 0;
18 }
```

Código 2: Operador -> para acessar struct.

Vetor

```
1
2 #include <stdlib.h>
3 #include <stdio.h>
4 int main(int argc, char const *argv[])
5 {
6     int *vetor = malloc(sizeof(int) * 10);
7     for (int i = 0; i < 10; i++)
8         scanf("%d", &vetor[i]);
9     for (int i = 0; i < 10; i++)
10         printf("[ %d ] ", vetor[i]);
11
12     free(vetor);
13     return 0;
14 }
```

Código 3: Exemplo de vetor dinamicamente alocado.

Matriz

Ponteiro para Ponteiro (double pointer)

- › Um ponteiro para ponteiro ($T **pp$) armazena o endereço de um ponteiro ($T *p$).
- › Exemplo para criar matrizes dinâmicas:
- › Acesso a $mat[i][j]$: $*(*(mat + i) + j)$ ou simplesmente $mat[i][j]$.

```
1 #include <stdlib.h>
2 #define LINHA 3
3 #define COLUNA 2
4 int main(int argc, char const *argv[])
5 {
6     int **mat;
7     mat = malloc(LINHA * sizeof(int*));
8     for(int i=0; i<LINHA; i++)
9         mat[i] = malloc(COLUNA * sizeof(int));
10    for (int i = 0; i < LINHA; i++)
11        free(mat[i]);
12    free(mat);
13    return 0;
14 }
```

Código 4: Exemplo de ponteiro para ponteiro.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #define LINHA 3
4 #define COLUNA 2
5 int main(int argc, char const *argv[]) {
6     int **matriz = malloc(sizeof(int*) * LINHA);
7     for (int i = 0; i < LINHA; i++) {
8         matriz[i] = malloc(sizeof(int) * COLUNA);
9     }
10    for (int i = 0; i < LINHA; i++)
11        for (int j = 0; j < COLUNA; j++)
12            scanf("%d", &matriz[i][j]);
13    for (int i = 0; i < LINHA; i++) {
14        for (int j = 0; j < COLUNA; j++) {
15            printf("[ %d ]", matriz[i][j]);
16        }
17        printf("\n");
18    }
19    for (int i = 0; i < LINHA; i++)
20        free(matriz[i]);
21    free(matriz);
22    return 0;
23 }
```

Código 5: Exemplo de matriz dinamicamente alocada.

Realocar

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(int argc, char const *argv[]) {
5     int tamanho = 10;
6     int *vetor = malloc(sizeof(int) * tamanho);
7
8     for (int i = 0; i < tamanho; i++)
9         scanf("%d", &vetor[i]);
10    for (int i = 0; i < tamanho; i++)
11        printf("[ %d ] ", vetor[i]);
12
13    tamanho++;
14    int *temp = realloc(vetor, sizeof(int) * tamanho);
15    vetor = temp;
16
17    scanf("%d", &vetor[tamanho - 1]);
18
19    for (int i = 0; i < tamanho; i++)
20        printf("[ %d ] ", vetor[i]);
21
22    free(vetor);
23    return 0;
24 }
```

Código 6: Exemplo de uso da função realloc().

Por que declarar vetor ou matriz como ponteiro é mais rápido?

- › **Acesso direto na heap:** ponteiros apontam para blocos contíguos na heap, sem necessidade de copiar dados da pilha.
- › **Passagem eficiente:** ao passar para funções, apenas o endereço é copiado, não todo o array.
- › **Redimensionamento com realloc:** evita alocar novo array e copiar manualmente.
- › **Menor uso de stack:** reduz risco de stack overflow em dados grandes.

(Deitel; Deitel, 2011) - Capítulo 12.3



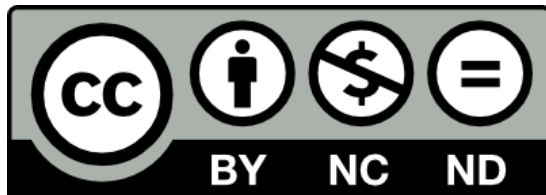


DEITEL, Paul; DEITEL, Harvey. **C: Como Programar**. 6. ed. São Paulo: Pearson Universidades, 2011.



SCHILDT, Herbert. **C Completo e Total: O Guia Definitivo para Programação em C**. 3. ed. São Paulo: Makron Books, 1996. ISBN 978-8534606928.

Estes slides estão protegidos por uma licença Creative Commons



Este modelo foi adaptado de Maxime Chupin.

Estrutura de Dados

Estrutura de Dados

Memória Dinamicamente Alocada