



# ESTRUTURAS DE DADOS EM LINGUAGEM C

# SUMÁRIO

- Estruturas de Dados
- Listas Simplesmente Encadeada
- Listas Duplamente Encadeada
- Filas
- Pilhas
- Grafos
- Árvores

ANDRÉ BACKES

# Estrutura de dados descomplicada em linguagem

C

# DEFINIÇÃO

Em computação, uma **estrutura de dados** é uma forma de armazenar e organizar os dados de modo que possam ser usados de forma eficiente.

Uma **estrutura de dados** é um relacionamento lógico entre diferentes tipos de dados visando à resolução de determinado problema de forma eficiente.

Trata-se de um tema fundamental da ciência da computação, pois a organização de forma coerente dos dados permite a diminuição do custo de execução de um algoritmo em termos de tempo de execução, consumo de memória ou ambos.

# ALOCAÇÃO DE MEMÓRIA: ESTÁTICA

Na alocação estática de memória, o programador não precisa se preocupar em reservar espaço de memória para seus dados (**variáveis, parâmetros, retorno de funções e endereço de outras**), que são automaticamente reservados na **pilha (stack)**.

Eles são armazenados sequencialmente em memória e a quantidade total de memória utilizada pelo programa é previamente conhecida e não pode ser alterada.

**Passo 2**

y = 3  
x = 2

stack

**Passo 3**

a = 2  
b = 3

y = 3  
x = 2

stack

**Passo 4**

c = 5  
a = 2  
b = 3

z  
y = 3  
x = 2

stack

**Passo 1**

x = 2

stack

```
int soma(int a, int b) { •  
    int c = a + b; •  
    return c;  
}  
int main() {  
    •int x = 2  
    •int y = 3;  
    •int z = soma(x, y); •  
    printf("soma = %d\n", z);  
    return 0;  
}
```

**Passo 5**

z = 5  
y = 3  
x = 2

stack

# ALOCAÇÃO DE MEMÓRIA: DINÂMICA

Neste caso, o programador tem total controle sobre o tamanho e tempo de vida das posições de memória dos seus dados.

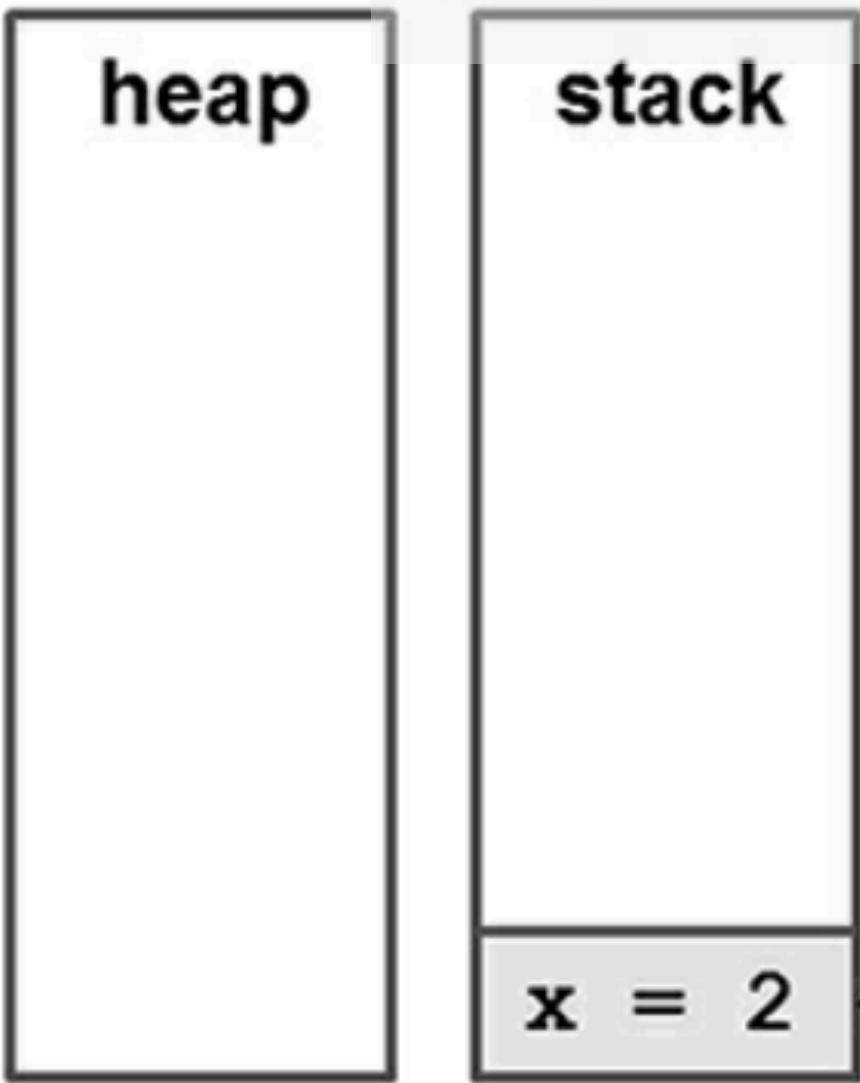
E não se reserva neste caso a memória na stack.

Reserva-se em outra área da memória: **heap**. É bem maior e mais lenta que a stack.

E estes espaços devem ser liberados manualmente pelo programador. Caso se esqueça, ocorre um vazamento de memória (**memory leak**).

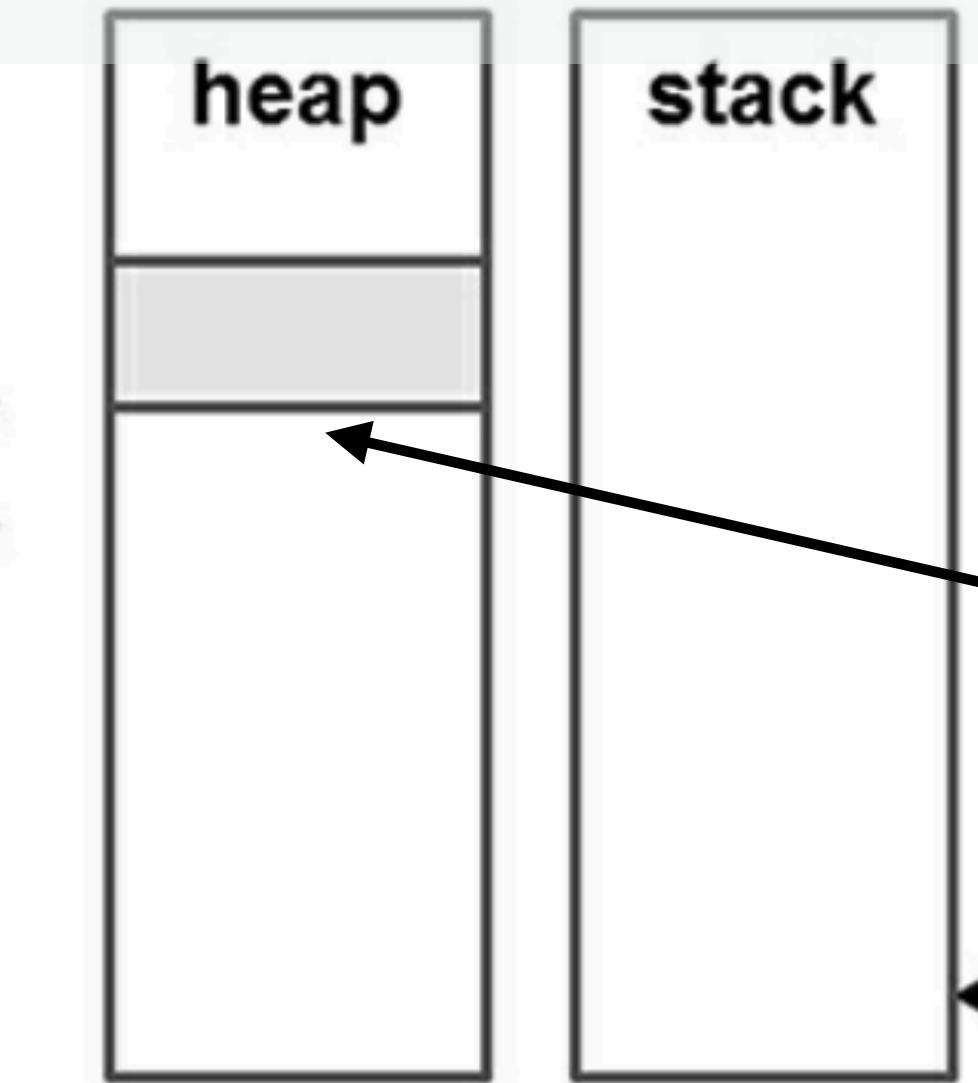
Em linguagem C, usa-se a função **malloc()** e só podem ser acessados por **ponteiros**. E os dados não precisam estarem sequenciais.

**Passo 1**

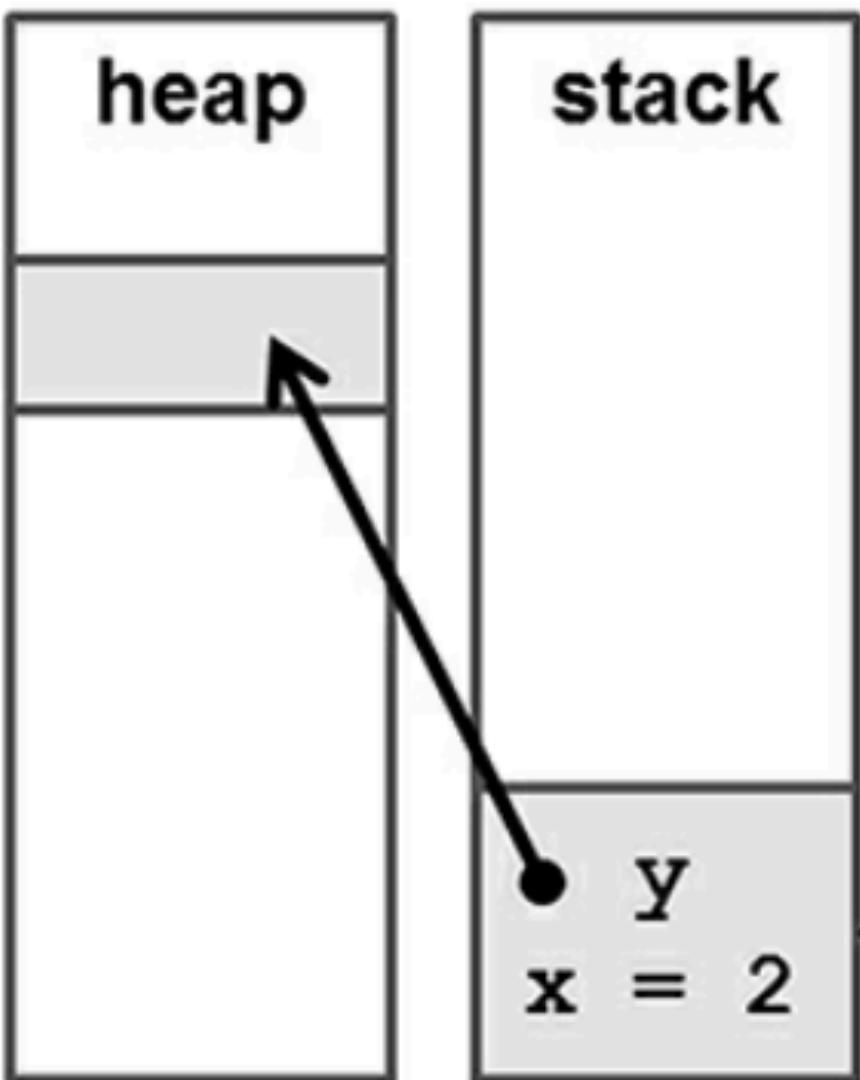


```
int main() {
    • int x = 2
    • int *y = malloc(100);
    • int *z = malloc(400);
    free(z);
    return 0;
}
```

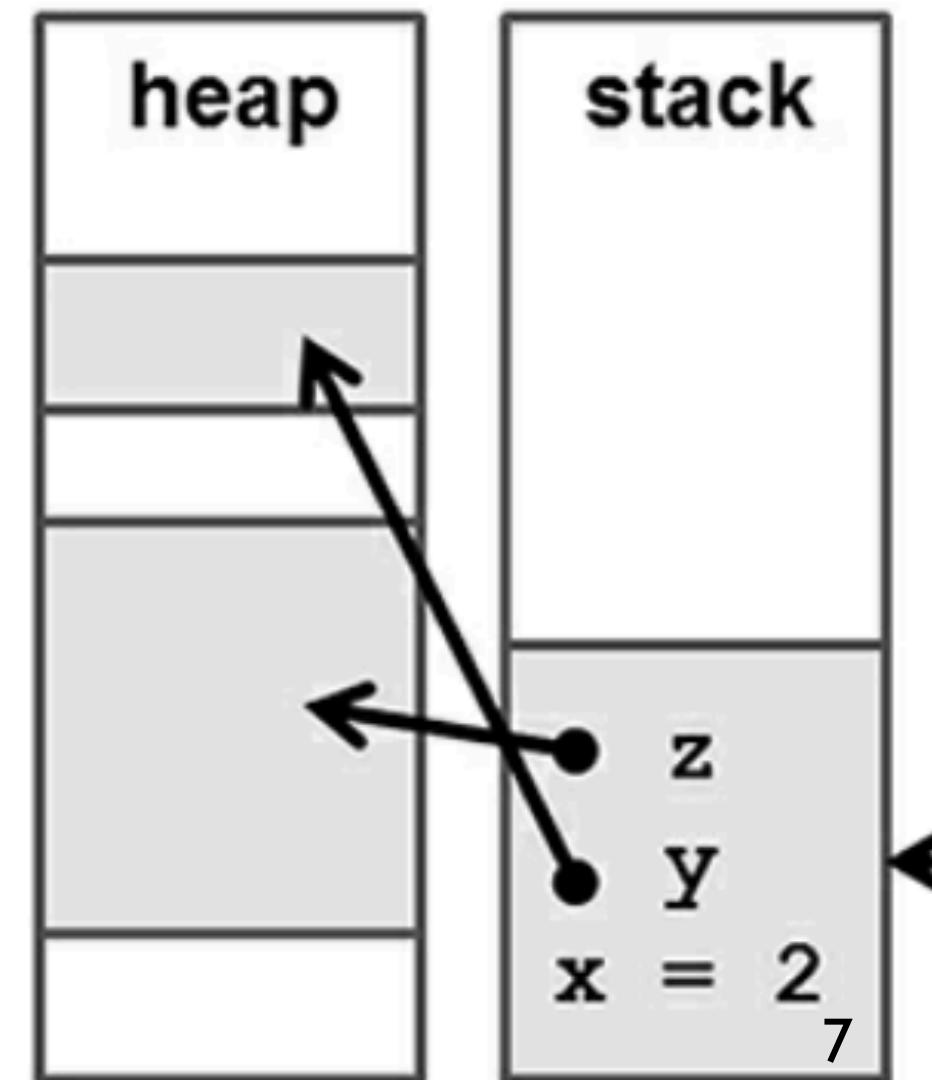
**Passo 5**



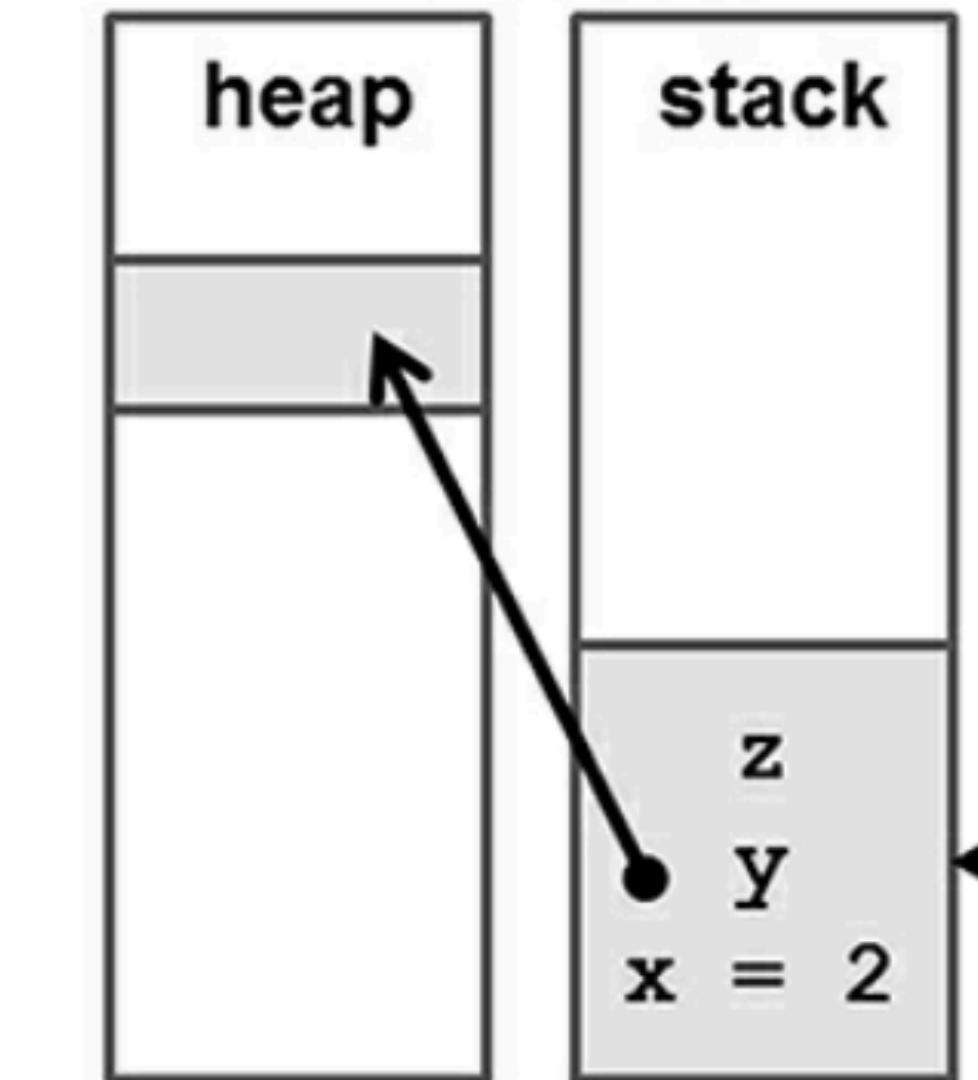
**Passo 2**



**Passo 3**



**Passo 4**



Alocação estática (stack)	Alocação dinâmica (heap)
Armazenado na memória RAM	Armazenado na memória RAM
Variáveis são liberadas automaticamente no final do escopo	Variáveis não dependem do escopo (podem ser acessadas globalmente) e devem ser liberadas manualmente
Alocação mais rápida que na heap	Alocação mais lenta que na stack
Implementado usando uma estrutura de dados do tipo pilha	Blocos de dados são alocados sob demanda
Armazena dados locais e endereços de retorno utilizados na passagem de parâmetros	Pode sofrer fragmentação após sucessivas alocações e liberações de memória
Os dados podem ser usados sem ponteiros	Os dados são acessados por ponteiros
Pode ocorrer estouro de pilha quando a pilha é muito usada	A alocação pode falhar se muita memória é solicitada
É usada quando se sabe exatamente o quanto de espaço será alocado antes do tempo de compilação e esse espaço não é muito grande	É usada quando não se sabe exatamente o quanto de espaço será alocado antes do tempo de compilação ou esse espaço é muito grande
Geralmente possui um tamanho máximo predeterminado quando o programa inicia	Responsável por vazamentos de memória

# TIPO ABSTRATO DE DADOS (TAD)

Um **tipo de dado** (**int**, **char**, **float**, **double**) define o conjunto de **valores** (domínio) e **operações** que uma variável deste tipo pode assumir.

Estrutura de dados é uma forma de organizar e armazenar os dados de modo que possam ser usados de forma eficiente: **array**, **struct**, **union**, **enum**.

Quando não existir uma forma eficiente de representar os dados que você precisa, deve ser criado um **TAD**.

Devemos separar o conteúdo de um TAD em dois arquivos:

- Arquivo .c com o conteúdo oculto do usuário (**implementação**).
- Arquivo .h com apenas a declaração das funções (**interface**).

# TAD: OPERAÇÕES BÁSICAS

- Criação do **TAD**.
- Inserção de um novo elemento no **TAD**.
- Remoção de um elemento do **TAD**.
- Acesso a um elemento do **TAD**.
- Destruição do **TAD**.

# EXEMPLO DE TAD: PONTO

Podemos criar um TAD que representa um ponto definido por suas coordenadas **x** e **y**.

Criaremos dois arquivos: **Ponto.h** e **Ponto.c**

Arquivo Ponto.h		Arquivo Ponto.c
01 <b>typedef struct</b> ponto Ponto; 02 //Cria um novo ponto 03 Ponto* Ponto_cria( <b>float</b> x, <b>float</b> y); 04 //Libera um ponto 05 <b>void</b> Ponto_libera(Ponto* p); 06 //Acessa os valores "x" e "y" de um ponto 07 <b>int</b> Ponto_acessa(Ponto* p, <b>float</b> * x, <b>float</b> * y); 08 //Atribui os valores "x" e "y" a um ponto 09 <b>int</b> Ponto_atribui(Ponto* p, <b>float</b> x, <b>float</b> y); 10 //Calcula a distância entre dois pontos 11 <b>float</b> Ponto_distancia(Ponto* p1, Ponto* p2);		01 <b>#include</b> <stdlib.h> 02 <b>#include</b> <math.h> 03 <b>#include</b> "Ponto.h" //inclui os Protótipos 04 <b>struct</b> ponto{//Definição do tipo de dados 05 <b>float</b> x; 06 <b>float</b> y; 07 };

# PONTO.C

## Criando um ponto

```
01 Ponto* Ponto_cria(float x, float y){  
02     Ponto* p = (Ponto*) malloc(sizeof(Ponto));  
03     if(p != NULL) {  
04         p->x = x;  
05         p->y = y;  
06     }  
07     return p;  
08 }
```

## Destruindo um ponto

```
01 void Ponto_libera(Ponto* p){  
02     free(p);  
03 }
```



Por que criar uma função para destruir o **TAD** sendo que tudo que precisamos fazer é chamar a função **free()**?

# PONTO.C

## Acessando o conteúdo de um ponto

```
01 int Ponto_acessa(Ponto* p, float* x, float* y) {  
02     if(p == NULL)  
03         return 0;  
04     *x = p->x;  
05     *y = p->y;  
06     return 1;  
07 }
```

## Atribuindo um valor ao ponto

```
01 int Ponto_atribui(Ponto* p, float x, float y) {  
02     if(p == NULL)  
03         return 0;  
04     p->x = x;  
05     p->y = y;  
06     return 1;  
07 }
```

## Calculando a distância entre dois pontos

```
01 float Ponto_distancia(Ponto* p1, Ponto* p2) {  
02     if(p1 == NULL || p2 == NULL)  
03         return -1;  
04     float dx = p1->x - p2->x;  
05     float dy = p1->y - p2->y;  
06     return sqrt(dx * dx + dy * dy);  
07 }
```

# TESTA\_PONTO.C

## Exemplo: utilizando o TAD ponto

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include "Ponto.h"
04 int main() {
05     float d;
06     Ponto *p, *q;
07     //Ponto r; //ERRO
08     p = Ponto_cria(10,21);
09     q = Ponto_cria(7,25);
10     //q->x = 2; //ERRO
11     d = Ponto_distancia(p,q);
12     printf("Distancia entre pontos: %f\n",d);
13     Ponto_libera(q);
14     Ponto_libera(p);
15     system("pause");
16     return 0;
17 }
```

# TAREFAS

- 1) Desenvolva um TAD que represente um cubo. Inclua as funções de inicializações necessárias e as operações que retornem os tamanhos de cada lado, a sua área e o seu volume.
- 2) Desenvolva um TAD que represente um cilindro. Inclua as funções de inicializações necessárias e as operações que retornem a sua altura e o raio, a sua área e o seu volume.
- 3) Desenvolva um TAD que represente uma esfera. Inclua as funções de inicializações necessárias e as operações que retornem o seu raio, a sua área e o seu volume.

# TAREFAS

4) Desenvolva um TAD que represente um número complexo  $z = x + iy$ , em que  $i^2 = -1$ , sendo  $x$  a sua parte real e  $y$  a parte imaginária. O TAD deverá conter as seguintes funções:

- Criar um número complexo.
- Destruir um número complexo.
- Soma de dois números complexos.
- Subtração de dois números complexos.
- Multiplicação de dois números complexos.
- Divisão de dois números complexos.



# OBRIGADO!

RAFAELVC2@GMAIL.COM