



Programação Imperativa e Funcional

Introdução a Ponteiros e Alocação Dinâmica

Diego Bezerra

dfb2@cesar.school



Nas aulas anteriores...



- Struct, Union e Enum
- Listas de exercícios
- Provas

Objetivos



- Introduzir
 - Ponteiros
 - Alocação dinâmica


Endereços



- Quais são as características da variável x, declarada a seguir?
 - Tipo: int
 - Nome: x
 - Endereço de memória ou referência: 0xbfd267c4
 - Valor: 9
- Para acessar o endereço de uma variável, utilizamos o operador &

```
int x = 9;  
printf("O endereço de memória de x é %p\n", &x);
```

Memória




5000	
5001	
5002	
5003	
5004	
...	
5007	

Endereços de memória Valor na memória

- A memória é formada por várias células
- Cada célula contém um endereço e um valor
- O tamanho do endereço e do valor dependem da arquitetura

Memória



5000	105
5001	0
5002	
5003	
5004	
...	
5007	

Endereços de memória

Valor na memória

- Exemplo
- O caractere i ocupa 1 byte na memória
- Inteiro d ocupa 2 bytes na memória

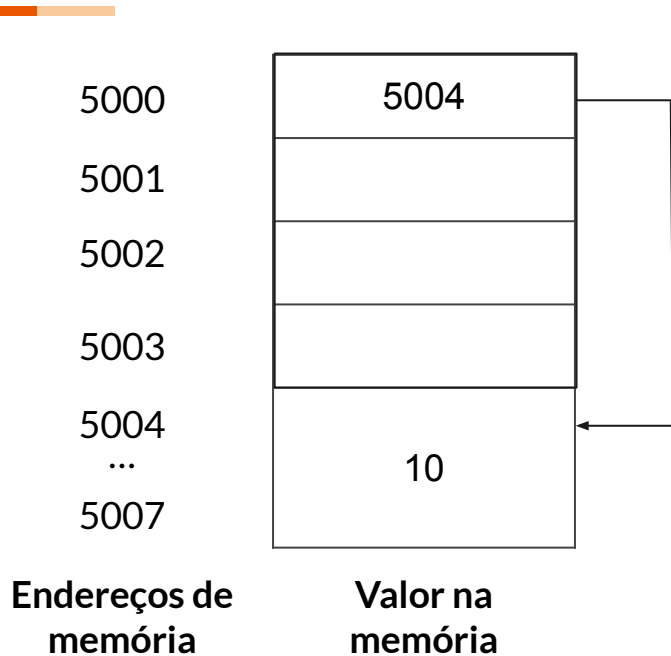
```
int main() {  
    char i = 'i';  
    int d = 0;  
    printf("%p", &i); // 5000  
    printf("%p", &d); // 5001  
    return 0;  
}
```

Ponteiros em C



- Variável especial que contém um **endereço de memória** armazenado
- A variável que contém esse endereço aponta para outra variável
 - Por meio do endereço de memória
- O **tipo apontado** pelo ponteiro deve ser declarado
- Pode apontar para variáveis de qualquer tipo, inclusive os tipos compostos e criados (ex.: struct, union e enum)

Ponteiros em C



Identificador	tipo	valor
ptrInput	int* (4 bytes)	5004
input	int(4 bytes)	10



Declarando ponteiros



- O tipo define para que tipo de variável o ponteiro está apontando
- Faz uso do operador de indireção (ou dereferência) *
- O uso do operador de endereço permite obter o endereço de memória de uma variável
- Importante! Recomenda-se sempre inicializar um ponteiro e verificar se seu valor não é **NULL (equivale a zero)**.

<tipo> <operador de indireção> <identificador>

```
//tipo *identificador;  
float *ptrMedia;  
float media = 8.5;  
ptrMedia = &media;
```

Ponteiros: operadores



Operador	Descrição
x	Valor armazenado pela variável
*x	Valor armazenado pela variável apontada pelo ponteiro
&x	Endereço de memória da variável

Declarando ponteiros



```
int main()
{
    int *x, valor, y;
    valor = 10;
    x = &valor; // Atribuindo o endereço de valor ao ponteiro x;
    y = *x; // Atribuindo o conteúdo da variável apontada por x a y;

    return 0;
}
```

Testando ponteiros



```
int main()
{
    int *x, valor, y;
    valor = 10;
    x = &valor; // Atribuindo o endereço de valor a x;
    y = *x; // Atribuindo o conteúdo da variável apontada por x a y;

    printf("%p", &valor); // Endereço da variável valor 0x7ffc9600b8c8
    printf("%p", x); // Endereço apontado pelo ponteiro 0x7ffc9600b8c8
    printf("%p", &x); // Endereço do ponteiro na memória 0x7ffc9600b8d0
    printf("%d", *x); // 10
    printf("%d", y); // 10

    return 0;
}
```

Exemplos: uso de ponteiros



```
int main()
{
    int *x, *y, valor;
    valor = 0;
    x = &valor; // Atribuindo o endereço de valor a x;
    y = &(*x);
    *y = 10;

    printf("%d", valor);

    return 0;
}
```

Exemplos: uso de ponteiros



```
int main()
{
    int *x, *y, valor;
    valor = 0;
    x = &valor; // Atribuindo o endereço de valor a x;
    y = *(&*x);

    printf("%d", valor);

    return 0;
}
```

Passagem por valor



- Qual o problema da função a seguir?
- Os parâmetros são passados por valor!
- Valores são passados para a e b
- A função efetua a troca dos valores entre as variáveis a e b?

```
void trocarValores(int a, int b) {  
    int aux = a;  
    a = b;  
    b = aux;  
}  
  
int main() {  
    int a = 2;  
    int b = 3;  
    trocarValores(a, b);  
    printf("%d %d", a, b);  
}
```

Passagem por valor



- Passagem de parâmetros por cópia de valor deve ser feita quando não queremos modificar os valores (variáveis) originais
- Forma simples e fácil de entender
- No entanto, computacionalmente custa caro
 - Maior consumo de memória e CPU
- Muitas vezes queremos que a alteração seja refletida em todo o programa (valor e variável original)
- Podemos então modificar o valor original usando **ponteiros**

Passagem de ponteiros



- C não implementa passagem por referência
- A solução é utilizar ponteiros para simular a passagem por referência
- A função recebe ponteiros para duas variáveis
- Em seguida, troca o conteúdo das memórias apontadas
- E no caso de vetores?

```
void trocarValores(int *a, int *b)
{
    int aux = *a;
    *a = *b;
    *b = aux;
}

int main() {
    int a = 2;
    int b = 3;
    trocarValores(&a, &b);
    printf("%d %d", a, b);
}
```

Exemplo de execução

```
void trocarValores1(int a, int b) {
    int aux = a;
    a = b;
    b = aux;
}

void trocarValores2(int *a, int *b) {
    int aux = *a;
    *a = *b;
    *b = aux;
}

int main() {
    int a = 2, b = 3;
    trocarValores1(a, b);
    printf("%d %d", a, b);
    trocarValores2(&a, &b);
    printf("%d %d", a, b);
}
```

Endereço	Conteúdo	Nome
0x1000	3	a
0x1004	2	b
0x1008		
0x1012		
0x1016		
0x1020		
0x1024		
0x1028		
0x1032		

Exercícios



- Indique o erro e proponha uma solução.

```
int main() {  
    int valor;  
    scanf("%d", &valor);  
  
    int *p = &valor;  
    *p = (*p) * (*p);  
    printf("Valor ao quadrado = %d\n", valor);  
    return 0;  
}
```

Ponteiros e vetores



- O nome de um vetor em C é, na verdade, um ponteiro que aponta para o primeiro elemento do vetor
- **vetor[i]** é equivalente a ***(vetor + i)**
- Ambos acessam o i-ésimo elemento do vetor.

```
int main() {  
    int vetor [3] = {10, 20, 30};  
    int *ptr = vetor;  
    printf("%d\n", *(ptr + 1));  
    return 0;  
}
```

Ponteiros e vetores



- Se uma string é um vetor de char, então...

```
#include <stdio.h>
#include <string.h>

int main() {
    char nome[] = "Larissa";
    char *ptrNome = nome;
    for(int i = 0; i < strlen(nome); i++){
        printf("%c\n", *(ptrNome + i));
    }
    return 0;
}
```

```
struct Pessoa {char *nome;};

void inicializar(Pessoa *p, char
*nome) {
    p->nome = nome;
}

int main() {
    char nome[] = "Larissa"
    Pessoa p = {"Rafael"};
    inicializar(&p, nome);
    return 0;
}
```

Ponteiros e structs



- Podemos acessar e modificar membros de forma eficiente
- Usamos o operador ‘->’ para acessar membros de uma struct quando temos ponteiros para ela

```
struct Pessoa { int altura; int idade; };

void preenchePessoa(struct Pessoa *p) { p->altura = 170; p->idade = 25; }

int main() {
    struct Pessoa p1;
    preenchePessoa(&p1); // Passando o endereço da struct
    printf("Altura: %d, Idade: %d\n", p1.altura, p1.idade); // Saída: 170,
    25
    return 0;
}
```

Exercícios



- Crie um procedimento que duplica o conteúdo da memória apontada por um ponteiro p. Crie um segundo procedimento que eleva o mesmo número a um valor n. Utilize os protótipos a seguir:

void duplica(int *p);

void potencia(int *p, int n);

- Faça um único procedimento que converte um valor em metros para: (i) jardas; (ii) pés; e (iii) polegadas. Use a função no método main(). Lembre-se que 1 metro é igual a aproximadamente 1.0940 jardas, 3.2810 pés, e 39.3701 polegadas.

Dica: utilize o protótipo: **void dist(float metros, float *jardas, float *pes, float *polegadas);**

Alocação dinâmica



- Permite **utilizar a memória de forma otimizada**
 - **Eficiência** no gerenciamento dos recursos
- Torna o uso das estruturas de dados **flexíveis**
 - Ex.: Vetor alocado **dinamicamente** vs **estaticamente**
- Resolução de **desafios mais complexos**
 - Listas, filas, pilhas e árvores (Algoritmos e Estruturas de Dados)
- Fundamento para programação avançada
 - Jogos, Sistemas Embarcados, Internet das Coisas

Operador malloc



- Possível fazer alocação dinâmica de memória usando o comando malloc
 - Faz parte da biblioteca <stdlib.h>
 - Aloca blocos consecutivos de bytes na memória e retorna o endereço de memória deste bloco

```
#include <stdlib.h>
int main() {
    double *ptrNumero = malloc(sizeof(double));
    *ptrNumero = 3.5;
    printf("Endereço de memória: %p\n", ptrNumero); // 0x7feaf4400690
    printf("Valor na memória: %lf\n", *ptrNumero); // 3.500000
    return 0;
}
```

Operador sizeof



- O operador permite saber o número de bytes ocupados por um determinado tipo de variável (que depende da arquitetura)
- Sintaxe: **sizeof(tipo)**
- Possível informar tanto o nome do tipo, uma variável ou ponteiro
- Retorna um valor inteiro que representa a quantidade de bytes

```
#include <stdlib.h>
int main() {
    double *ptrNumero = malloc(sizeof(double));
    *ptrNumero = 3.5;
    printf("Endereço de memória: %p\n", ptrNumero); // 0x7feaf4400690
    printf("Valor na memória: %lf\n", *ptrNumero); // 3.500000
    return 0;
}
```

Definindo cadeias usando malloc

- Usando malloc para criar um bloco com 3 doubles

```
#include <stdlib.h>

int main() {
    int n;
    scanf("%d", &n);
    double *ptrNumeros = malloc(n * sizeof(double));
    ptrNumeros[0] = 1.1;
    ptrNumeros[1] = 1.2;
    ptrNumeros[2] = 2.2;

    for (int i = 0; i < n; i++) { printf("%.11f ", ptrNumeros[i]); }
    // imprime: 1.1 1.2 2.2
    return 0;
}
```

Alocando cadeias dinamicamente

```
#include <stdlib.h>

int main() {

    int *a = NULL;
    a = malloc(6 * sizeof(int));
    for (int i = 0; i < 6; i++)
        a[i] = i*2;
    imprimeVetor3(a, 6);
}
```

- Usando malloc para criar um bloco com 6 doubles
- Armazena o endereço de memória no ponteiro a
- Altera o valor de cada posição

Endereço	Conteúdo	Nome
0x1000	0x1008	a
0x1004		
0x1008	0	Vetor dinâmico a
0x1012	2	
0x1016	4	
0x1020	6	
0x1024	8	
0x1028	10	
0x1032		

Alocando uma struct dinamicamente



- Uma **struct** representa um tipo composto, logo também é possível alocar dinamicamente uma variável desse tipo

```
#include <stdlib.h>
struct data {int dia, mes ano;} typedef Data;
int main() {
    Data *d = malloc(sizeof(Data));
    d->dia = 01; d->mes = 10; d->ano = 2024;
    return 0;
}
```

Exercícios



- Escreva uma função que receba um `int c` (que pode representar um caractere ASCII, por exemplo) e transforme-o em uma string, ou seja, devolva uma string de comprimento 1 tendo `c` como único elemento. Lembre-se de usar o `'\0'`
- Discuta, passo a passo, o efeito do seguinte fragmento de código:

```
int *v = malloc (10 * sizeof (int));
```

- Discuta o efeito do seguinte fragmento de código:

```
int *x = malloc (10 * sizeof *x);
```

Liberando memória alocada



- A memória alocada de forma estática pelo compilador é liberada automaticamente
- Quando alocamos dinamicamente, é **nossa responsabilidade liberar o espaço de memória usado**
- Em C, usamos o procedimento **free**

```
#include <stdlib.h>
int main() {

    int *a = malloc(sizeof(int));
    free(a);
    return 0;
}
```

Redimensionamento e a função realloc



- A função **realloc** aumenta o tamanho de um bloco de memória que tenha sido alocado dinamicamente
- Também pode diminuir o tamanho de um bloco de memória alocado dinamicamente
- A função aloca um bloco do tamanho desejado, copia para ele o conteúdo do bloco original, e devolve o endereço do novo bloco. A função libera o bloco original invocando **free**

Redimensionamento e a função realloc



```
#include <stdlib.h>
int main() {
    int *v;
    v = malloc (1000 * sizeof (int));
    v = realloc (v, 2000 * sizeof (int));
    return 0;
}
```

A memória é finita



- Se a memória do computador já estiver toda ocupada, malloc não consegue alocar mais espaço e devolve **NULL**
- Quando alocamos dinamicamente, é nossa **responsabilidade liberar o espaço de memória usado**
- Em C, usamos o procedimento **free**

```
#include <stdlib.h>
int main() {

    int *a = malloc(sizeof(int));
    free(a);
    return 0;
}
```

```
#include <stdlib.h>
#include <stdio.h>

void lerVetor(int *v, int n) { for (int i = 0; i < n; i++) scanf("%d",
&v[i]);}

int *maior(int *v, int n) {
    int *maior = v;
    for (int i = 1; i < n; i++) { if (v[i] > *maior) {maior = v + i;}}
    return maior;
}

int main() {
    int n, *v;
    scanf("%d", &n);
    v = malloc(n * sizeof(int));
    int *valor = maior(v, n);
    printf("Maior valor: %d", *valor);
    free(v);
    return 0;
}
```

Exercícios



- Crie uma função que:
 - Recebe um vetor `v` e seu tamanho `n` por parâmetro;
 - Cria um novo vetor por alocação dinâmica, preenchendo-o com o conteúdo de `v` em ordem inversa
 - **Retorne o novo vetor**
 - Use o protótipo `int *inverter(int *v, int n);`
 - Crie um exemplo de uso desta função na função `main()`
 - Não esqueça de liberar a memória utilizada!

Exercícios



- Crie um programa que gerencie o cadastro de estudantes de uma escola. O programa deve permitir adicionar estudantes em um vetor e listar os cadastrados. O uso de funções e ponteiros é obrigatório.
- Armazene em um vetor todos os estudantes para organizar os dados.
- Cada estudante deve ter as seguintes informações:
 - Nome (string)
 - Idade (int)
 - Nota (float)

Sugestão de leitura



- Capítulo 5 do livro de W. Celes e J. L. Rangel (2004)
 - Vetores e alocação dinâmica: página 5-8



Referências



- Rangel Netto, J. L. M., Cerqueira, R. D. G., & Celes Filho, W. (2004). **Introdução a estrutura de dados: com técnicas de programação em C.**