Ponteiros

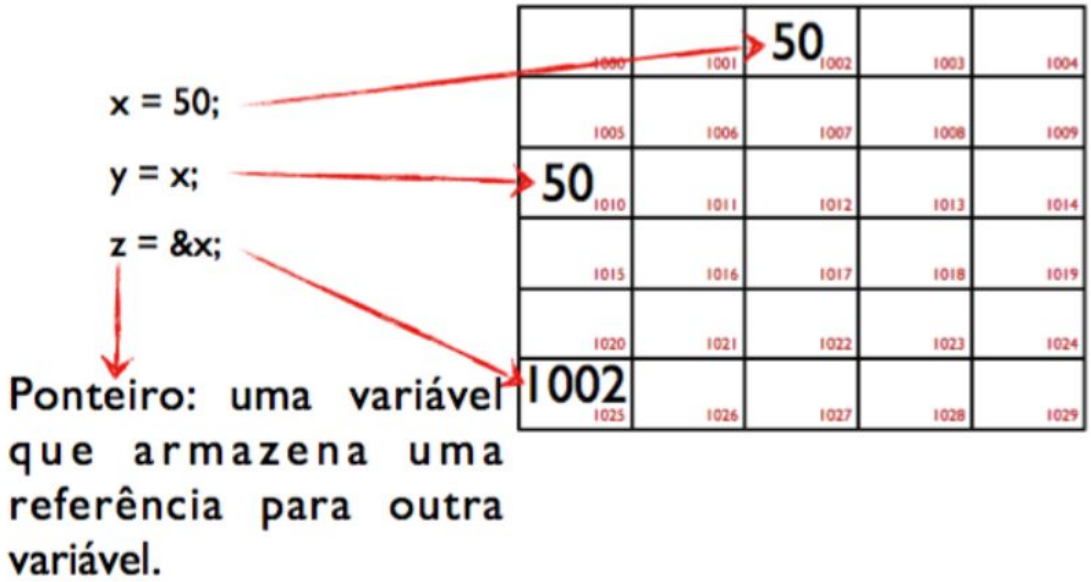
Alocação Dinâmica de Memória

Sumário

- Ponteiros
- Alocação dinâmica de Memória

Ponteiro

- Um ponteiro é um tipo especial de variável que armazena não um valor propriamente dito, e sim o endereço de memória onde o valor está sendo armazenado



Ponteiro

- Declaração de um ponteiro:

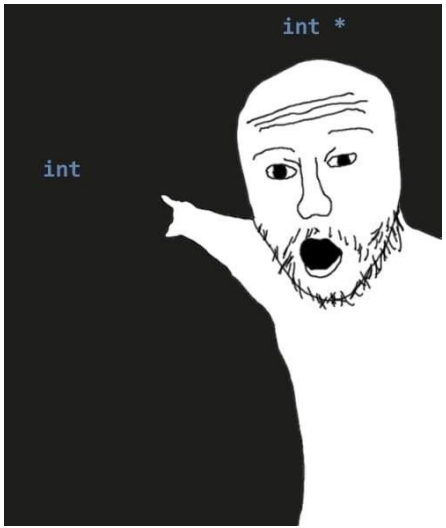
```
tipo *nomePonteiro;
```

- Atribuição de endereço:

```
nomePonteiro = &varNormal;
```

- Declaração + atribuição:

```
tipo *nomePonteiro = &varNormal;
```



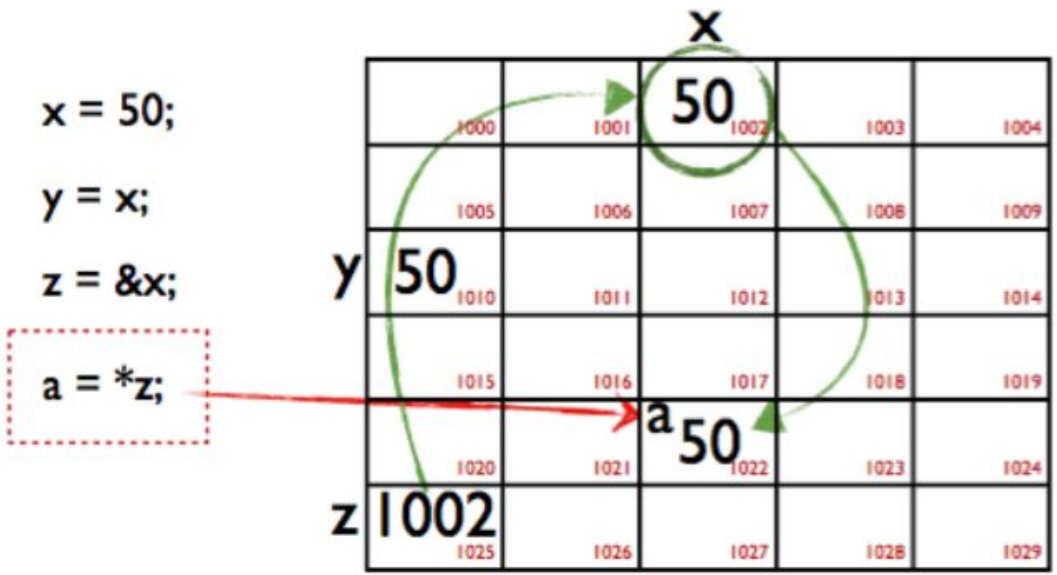
Ponteiro

- Acesso ao valor apontado pelo ponteiro:

  varNormal = *nomePonteiro

- Alteração do valor apontado pelo ponteiro:

  *nomePonteiro = novoValor



Ponteiro

- Exemplo:

```c
#include <stdio.h>
int main() {
  int a = 8;
  int *p;
  p = &a;

  printf("Valor de a: %d\n", a);
  printf("Endereco de a: %p\n", &a);
  printf("Valor de p: %p\n", p);
  printf("Valor apontado por p: %d\n", *p);
  return 0;
}
```

Mesmo valor

Mesmo endereço

Ponteiro

- Exemplo:

```c
#include <stdio.h>
int main() {
  int a = 8;
  int *p;
  p = &a;
  *p = 21;
  printf("Valor de a: %d\n", a);
  return 0;
}
```

Está alterando o valor da variável **a**

Ponteiro

- Ponteiro não inicializado:
  - aponta para um local indefinido da memória, o qual pode conter lixo

```
int *p;
```

- Valor especial NULL:
  - Sinaliza que o ponteiro não aponta para um local específico da memória

```
int *p = NULL;
```

Ponteiro para Struct

```c
struct data {
  int dia;
  int mes;
  int ano;
};
```

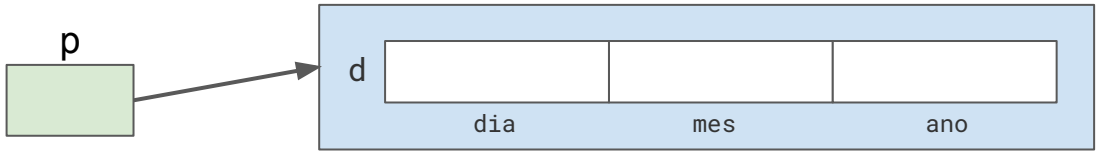
- A declaração e vinculação de ponteiros para tipos definidos através de struct obedece ao mesmo formato já visto:

```c
struct data d;
```

Variável **d** do tipo struct data

```c
struct data *p = &d;
```

Ponteiro **p** para **d**



Ponteiro para Struct

- Como acessar os campos da struct através do ponteiro?

```
*p.dia
*p.mes
*p.ano
```

```
(*p).dia
(*p).mes
(*p).ano
```

ou

```
p->dia
p->mes
p->ano
```

Exercício

1. Escreva um programa que declare um inteiro, um float e um char, além de ponteiros para inteiro, float e char. Associe as variáveis aos ponteiros usando &. Modifique os valores de cada variável através de seus respectivos ponteiros. Imprima os valores das variáveis antes e após a modificação.

2. Escreva um programa que declare uma struct para armazenar o nome e o preço de um produto. No main, crie uma variável deste tipo e um ponteiro para ela. Efetue a leitura e, através do ponteiro, a atribuição e a impressão dos dados de um produto.

Passagem de Parâmetros

- Quando passamos valores para uma função, podemos fazê-lo de duas formas:
  - Por valor (ou por cópia)
  - Por referência

Passagem de Parâmetros

- Quando passamos valores para uma função, podemos fazê-lo de duas formas:
  - **Por valor (ou por cópia)**
    - É a forma padrão
    - Os parâmetros da função recebem uma cópia das variáveis que foram passadas no ato da chamada à função
    - Qualquer modificação feita pela função incide sobre a cópia, ou seja, os valores das variáveis que foram passadas não se alteram

Passagem de Parâmetros

- Por valor:

```c
#include <stdio.h>

void desconto(int x){
  // Aqui a passagem é por valor
  x = x - 5;
  printf("Valor dentro da função: %d\n", x);

}
int main() {
  int var = 23;
  printf("Valor de var antes da função: %d\n", var);
  desconto(var);
  printf("Valor de var após a função: %d\n", var);
  return 0;
}
```

x recebe uma cópia do valor de **var**

var continua valendo 23

Passagem de Parâmetros

- Quando passamos valores para uma função, podemos fazê-lo de duas formas:
  - Por referência
    - O endereço de memória que armazena o dado é passado para a função
    - No escopo da função, o endereço é usado para acessar o dado real utilizado na chamada
    - Qualquer alteração é feita diretamente no endereço do dado e, portanto, será visível fora da função
    - Em C, a passagem por referência é implementada com o uso de ponteiros

Passagem de Parâmetros

- Por referência:

```c
#include <stdio.h>

void desconto(int *x){
  // Aqui a passagem é por referência
  *x = *x - 5;
  printf("Valor dentro da função: %d\n", *x);

}
int main() {
  int var = 23;
  printf("Valor de var antes da função: %d\n", var);
  desconto(&var);
  printf("Valor de var após a função: %d\n", var);
  return 0;
}
```

x recebe o endereço de **var**

passamos o endereço de **var**

var agora vale 18

Passagem de Parâmetros

- Algo familiar?

```
int num;
scanf("%d", &num);
```

- Não, pera…

```
char texto[10];
scanf("%s", texto);
```

?

Arrays (vetores, matrizes, strings)

- Em C, o nome de um array (quando utilizado sem o indicativo de índice) é um ponteiro para seu primeiro elemento
  - `nomeVetor`, `&nomeVetor` e `&nomeVetor[0]` se referem ao mesmo endereço na memória

- Arrays são organizados na memória de forma sequencial
  - Sabendo o endereço da primeira posição, conseguimos acessar todas as demais

Arrays como parâmetros de funções

- Em C, arrays são sempre passados por referência
  - Quando um array é usado como argumento de função, a função recebe o endereço do primeiro elemento
- As declarações abaixo são equivalentes:

```
void imprimeVetor(int *vet)

void imprimeVetor(int vet[])

void imprimeVetor(int vet[10])
```

Mesmo que seja informado o tamanho, ele será ignorado. No caso de matriz, deve-se informar apenas o número de colunas

Arrays como parâmetros de funções

- Se o tamanho do array não for conhecido pela função, pode ser necessário passar um parâmetro adicional que indique o número de elementos
  - Lembre-se que C não fornece verificação dos limites de vetores/matrizes

```
void imprimeVetor(int *vet, int tam)
```

Arrays como parâmetros de funções

- Exemplo

```
void imprimeVetor(int vet[], int tam){
  for(int i = 0; i < tam; i++){
    printf("%d ", vet[i]);
  }
}
```

Como chamar:

```
imprimeVetor(v, 5);
```

Não vai **&**, pois o nome já é o endereço do primeiro elemento

Exercício

3. Faça uma função que receba um vetor de 5 elementos e substitua cada valor pelo seu quadrado. No main, chame a função **imprimeVetor** antes e após a alteração dos valores.

Sumário

- Ponteiros
- **Alocação Dinâmica de Memória**

Alocação Dinâmica de Memória

- Até o momento, trabalhamos com variáveis cujo espaço na memória é alocado de forma **estática**
  - O espaço para as variáveis globais é alocado em tempo de compilação
  - O espaço para as variáveis locais/parâmetros é alocado na pilha no momento da execução da função
- Isso funciona bem quando sabemos exatamente de quanto espaço iremos precisar, já que variáveis globais ou locais não podem ser acrescentadas em tempo de execução (*runtime*)
- E se precisarmos armazenar um conjunto de dados cujo número de elementos ainda é desconhecido?

Alocação Dinâmica de Memória

- Alocação dinâmica é o meio pelo qual um programa pode obter/liberar memória durante sua execução
- Em C, a utilização de ponteiros fornece suporte à utilização da alocação dinâmica
- As duas principais funções para alocação dinâmica, disponíveis na biblioteca `<stdlib.h>`, são:

malloc()	free()
Aloca memória	Libera memória

Alocação Dinâmica de Memória

- **malloc()**
  - Solicita um novo bloco de memória
  - Recebe por parâmetro a quantidade de bytes a ser alocada
  - Retorna:
    - NULL, em caso de erro
    - Um ponteiro para o início do espaço recém alocado, em caso de sucesso

  - Exemplos:
    - Alocar espaço para um vetor de 10 inteiros (40 bytes):

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

    - Alocar espaço para uma string com 20 caracteres + \0 (21 bytes):

```
char *s = malloc(21 * sizeof(char));
```

Alocação Dinâmica de Memória

- **free()**
  - Libera um espaço de memória dinamicamente alocada
  - Recebe por parâmetro o ponteiro para a memória a ser liberada

  - Exemplo:

```
free(v);
```

Alocação Dinâmica

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

int main(){
    int *v = malloc(10 * sizeof(int));
    if(v == NULL){
        printf("Memória insuficiente!");
        exit(1); // encerra o programa
    }

    int i, soma = 0;
    for(i = 0; i < 10; i++){
        printf("elemento v[%d]: ", i);
        scanf("%d", &v[i]);
        soma += v[i];
    }
    printf("Soma dos elementos: %d", soma);
    free(v);
    return 0;
}
```

Alocação Dinâmica de Memória

- Outras funções
  - **calloc()**
    - Recebe o número de elementos a serem alocados e o tamanho de cada um
    - Mesmo objetivo de malloc, com a diferença de que inicializa todos os bits do espaço alocado com zeros

```
int *v = calloc(10, sizeof(int));
```

  - **realloc()**
    - Realoca memória durante a execução do programa
    - Recebe um ponteiro para a memória anteriormente alocada e a nova quantidade de bytes

```
v = realloc(v, 5 * sizeof(int));
```

Exercício

- Repita estes exercícios, porém, sem declarar as variáveis estáticas. Em vez disso, a memória para os dados deve ser alocada dinamicamente e vinculada aos respectivos ponteiros.