# Ponteiros e suas aplicações

Victor Turrisi

# Introdução

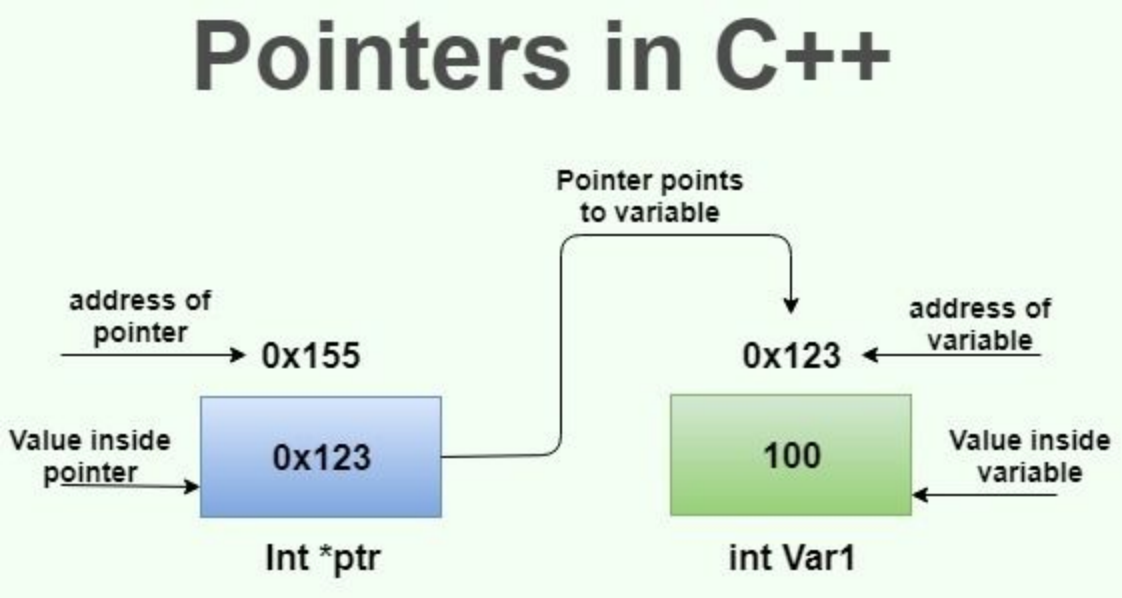
- Ponteiro são ferramentas extremamente importantes em qualquer linguagem de programação

- Mesmo linguagens que não “fornecem” ponteiros (por exemplo Python) os utilizam fortemente para diversas tarefas

# Conceitos

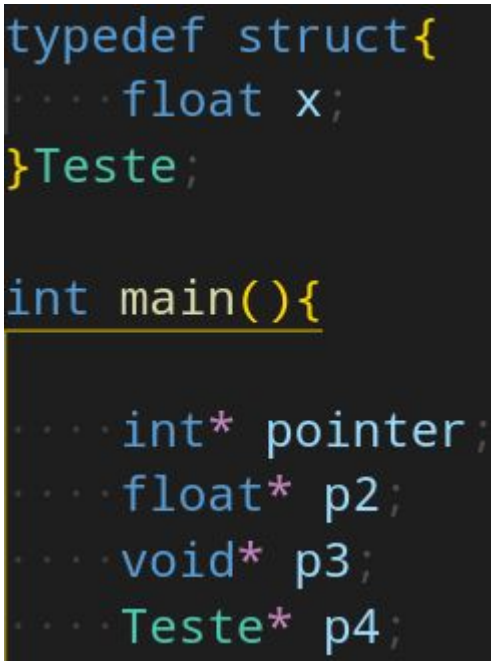
- Consiste em uma **referência** para algum objeto ou variável do seu código
- Na prática, um ponteiro não contém nenhuma informação relevante, atuando como um localizador de uma variável
- Se pensarmos na memória como um mapa com latitude e longitude, a UEL está localizada em uma determinada coordenada (endereço na memória) e contém diversas informações relevantes, e.g., nome, alunos, cursos, etc
- Um ponteiro para a UEL seria algo que sabe em quais coordenadas a UEL se encontra e não propriamente contém as informações da UEL

# Conceitos


Pointers in C++
Pointer points to variable
address of pointer
0x155
address of variable
0x123
Value inside pointer
0x123
100
Value inside variable
Int *ptr
int Var1

# Ponteiros em C

- Sintaxe
  - tipo da variável - * - nome da variável



# Ponteiros em C

- Após definirmos um ponteiro, devemos fazer ele referenciar alguma variável

```c
int c = 10;

printf("Endereço c: %p\n", &c);
printf("Valor c: %d\n\n", c);

printf("Endereço pointer antes de referenciar: %p\n", &pointer);
printf("Valor pointer antes de referenciar: %p\n\n", pointer);

pointer = &c; // fazemos o ponteiro referenciar c

printf("Endereço c: %p\n", &c);
printf("Valor c: %d\n\n", c);

printf("Endereço pointer depois de referenciar: %p\n", &pointer);
printf("Valor pointer depois de referenciar: %p\n\n", pointer);
```

# Ponteiros em C

- Podemos ver que após referenciar a variável c, o ponteiro pointer agora possui um outro valor, no caso, exatamente o endereço de memória de c

```
Endereço c: 0x7fff8371a4fc
Valor c: 10

Endereço pointer antes de referenciar: 0x7fff8371a500
Valor pointer antes de referenciar: 0x7fff8371a5f0

Endereço c: 0x7fff8371a4fc
Valor c: 10

Endereço pointer depois de referenciar: 0x7fff8371a500
Valor pointer depois de referenciar: 0x7fff8371a4fc
```

# Ponteiros em C

- Agora que o ponteiro já referência a variável desejada, temos que conseguir acessar o valor da variável c

- Isso é feito através da seguinte sintaxe

```
printf("Valor de c (utilizando pointer): %d\n\n", *pointer);
```

```
Valor de c (utilizando pointer): 10
```

# Ponteiros em C

- Portanto, para utilizarmos ponteiros, devemos:
  - Definir um ponteiro seguindo o formato tipo_da_variável_de_interesse * nome
    - ```
      int* pointer;
      ```
  - Referenciar a variável de interesse utilizando a sintaxe p = &variável
    - ```
      pointer = &c; // fazemos o ponteiro referenciar c
      ```
  - Acessar o valor da variável (quando desejado) utilizando o símbolo *
    - ```
      printf("Valor de c (utilizando pointer): %d\n\n", *pointer);
      ```
      ```

# Ponteiros

- Até o momento vimos como definir e utilizar ponteiros, mas qual a razão de usá-los?

# Ponteiros

- Até o momento vimos como definir e utilizar ponteiros, mas qual a razão de usá-los?

- Várias! Alguns casos:
  - passagem por valor x passagem por referência
  - vetores e matrizes

# Passagem por valor

- Forma de se passar parâmetros para uma função
- É geralmente o tipo visto primeiro
- Ele obriga que variáveis passadas por parâmetro devam ser **copiadas**

```c
// passagem por valor de x
int f_valor(int x) {
    printf("Endereço x: %p\n", &x);
    printf("Valor x: %d\n\n", x);
}
```

```c
f_valor(c);
```

```
Endereço c: 0x7ffd3f93ed74
Valor c: 10

Endereço x: 0x7ffd3f93ed5c
Valor x: 10
```

# Passagem por valor

- Desvantagens:
  - Custo de memória em dobro
  - Não conseguimos alterar o valor da variável original

- Vantagens:
  - Simplicidade
  - Não alteramos o valor da variável original

# Passagem por valor

```c
// passagem por valor de x
int f_valor(int x) {
    printf("Endereço x: %p\n", &x);
    printf("Valor x: %d\n", x);
    x = 30;
    printf("Valor x: %d\n", x);
}
```

```
Endereço c: 0x7ffc2d825744
Valor c: 10

Endereço x: 0x7ffc2d82572c
Valor x: 10
Valor x: 30
Valor c: 10
```

# Passagem por referência

- Consiste em passar um endereço de uma variável para uma função

- Essa função cria um ponteiro que referencia a variável passada

```c
int f_referencia(int* x){
    printf("Endereço x: %p\n", &x);
    printf("Valor x: %p\n\n", x);
    printf("Valor da variável referenciada por x: %d\n\n", *x);
    *x = 30;
}
```

```c
f_referencia(&c);
```

```
Endereço c: 0x7ffd37feff84
Valor c: 10
Endereço x: 0x7ffd37feff68
Valor x: 0x7ffd37feff84
Valor da variável referenciada por x: 10
Valor c (após chamar f_referencia): 30
```

# Passagem por valor

- Desvantagens:
  - Complexidade adicional
  - Alterações da variável dentro da função alteram a variável externa (Diferença de função x procedimento)

- Vantagens:
  - Custo de memória
  - Criar funções que alteram os estados de nosso código
    - Por exemplo, sem ponteiros, não seria possível criar uma função que ordena um vetor sem criar um outro vetor, realizar uma cópia e retornar o novo vetor

# Vetores e Matrizes

- Vamos supor um vetor criado manualmente

```c
int vetor[10];
vetor[0] = 1;
vetor[1] = 2;
vetor[2] = 3;

printf("Valor na posição 1 do vetor %d\n\n", vetor[1]);
```

- Internamente, como é feito o acesso à posição 0, 1 ou 2 do vetor?

# Vetores e Matrizes (com ponteiros)

- Vamos supor um vetor criado manualmente

```c
int vetor[10];
vetor[0] = 1;
vetor[1] = 2;
vetor[2] = 3;

printf("Valor na posição 1 do vetor %d\n\n", vetor[1]);
```

- Internamente, como é feito o acesso à posição 0, 1 ou 2 do vetor? Ponteiros!

# Vetores e Matrizes (com ponteiros)

```c
int vetor[10];
vetor[0] = 1;
vetor[1] = 2;
vetor[2] = 3;

printf("Valor na posição 1 do vetor %d\n\n", vetor[1]);

int* p;

p = vetor;
printf("%d (%p) (em inteiro %ld)\n", *p, p, p);
printf("%d (%p) (em inteiro %ld)\n", *(p+1), (p+1), (p+1));
```

```
Valor na posição 1 do vetor 2

1 (0x7ffdd05bd050) (em inteiro 140728099131472)
2 (0x7ffdd05bd054) (em inteiro 140728099131476)
```

# Vetores e Matrizes (com ponteiros)

```c
double vetor[] = {1.0, 2.1, 3.7, 4.19};

printf("Valor na posição 1 do vetor %lf\n\n", vetor[1]);

double* p;

p = vetor;
printf("%lf (%p) (em inteiro %ld)\n", *p, p, p);
printf("%lf (%p) (em inteiro %ld)\n", *(p+1), (p+1), (p+1));
```

```
Valor na posição 1 do vetor 2.100000

1.000000 (0x7ffc15ad18b0) (em inteiro 140720672151728)
2.100000 (0x7ffc15ad18b8) (em inteiro 140720672151736)
```

# Vetores e Matrizes (com ponteiros)

```c
int f_vetor(int* vetor){
    printf("Vetor: %d\n", vetor[2]);
}
```

```c
int vetor[] = {1, 2, 3, 4};
f_vetor(vetor);
```

```
Vetor: 3
```

# Vetores e Matrizes (com ponteiros)

```c
double vetor[] = {1.0, 2.1, 3.7, 4.19};

void* p;

p = vetor;

printf("Valor na posição 1 do vetor %lf (%p) (em inteiro %ld)\n",
        vetor[1], &vetor[1], &vetor[1]);
printf(
    "%lf (%p) (em inteiro %ld)\n",
    *(double *)(p+sizeof(double)),
    (p+sizeof(double)), (p+sizeof(double))
);
```

```
Valor na posição 1 do vetor 2.100000 (0x7ffd427a0db8) (em inteiro 140725718748600)
2.100000 (0x7ffd427a0db8) (em inteiro 140725718748600)
```

# Vetores e Matrizes (com ponteiros)

```c
int f_matriz(int (*matriz)[4]){
    printf("Matriz: %d (começo da segunda linha %p)\n", **(matriz+1), (matriz+1));
}
```

```c
int matriz[][4] = {{1, 2, 3, 4}, {4,5,6,7}};
printf("Endereço primeira linha: %p\n", &matriz[0]);
printf("Endereço segunda linha: %p\n", &matriz[1]);
f_matriz(matriz);
```

```
Endereço primeira linha: 0x7fffafa0e8d0
Endereço segunda linha: 0x7fffafa0e8e0
Matriz: 4 (começo da segunda linha 0x7fffafa0e8e0)
```

# Exercícios (parte 1)

1. Implemente um programa que leia 25 valores (double) fornecidos por um usuário e atribua esses valores em um vetor 1x25. Em seguida, receba um valor (int), i, de um usuário e imprima o valor correspondente a posição [i] na vetor

2. Modifique o programa criado para exercício anterior para que o acesso a posição [i] seja feita por meio de ponteiros do tipo void (utilizando o incremento de endereços visto em sala)

3. Crie uma função separada da main para realizar a operação de acesso ao índice (ela deve receber três parâmetros: vetor, i, j)

# Exercícios (parte 2)

1. Implemente um programa que leia 25 valores (double) fornecidos por um usuário e atribua esses valores em uma matriz 5x5. Em seguida, receba dois valores (int), i e j, de um usuário e imprima o valor correspondente a posição [i][j] na matriz

2. Modifique o programa criado para exercício anterior para que o acesso a posição [i][j] seja feita por meio de ponteiros do tipo void (utilizando o incremento de endereços visto em sala)

3. Crie uma função separada da main para realizar a operação de acesso ao índice (ela deve receber três parâmetros: matriz, i, j)