

PONTEIROS

Prof. Muriel Mazzetto
Algoritmos 2

Variáveis e ponteiros

2

- **Variáveis** armazenam dados:
 - ▣ int, float, double, char, struct;
- **Ponteiros** armazenam endereços de memória dos tipos de dados.
- Cada ponteiro só poderá armazenar o endereço do seu respectivo tipo de dado.

O que são ponteiros?

3

- ❑ Um **ponteiro** é uma variável que contém um endereço de memória.
- ❑ Esse endereço é normalmente a posição de uma outra variável na memória.
- ❑ Se uma variável contém o endereço de outra, então a primeira **aponta** para a segunda.

Declaração de ponteiro

4

- Os ponteiros são similares aos **tipos de dados**.
- Utiliza-se o operador `*` para determinar que a variável se trata de um **ponteiro do tipo especificado**.
- Tipo `*nome;`
`int *ponteiro_int;`
`char *ponteiro_char;`
`double *ponteiro_double;`

Operadores

5

- **&** - “o endereço de”
 - ▣ Devolve o endereço de memória do operando.
 - ▣ Exemplo: `m = &count;`
 - m recebe o endereço de count.
 - ▣ O endereço é a posição da variável na memória do computador.

Operadores

6

- * - “no endereço” ou “valor de”
 - ▣ Declarar variável do tipo ponteiro;
 - ▣ Acessa o valor armazenado no endereço apontado.
 - ▣ Exemplo: $q = *m$;
 - q recebe o valor que está no endereço m.

Operadores

7

```
int main(void)
{
    int var;
    int *p_var; //ponteiro de int

    var = 50;
    p_var = &var; //recebe ENDEREÇO DE var

    printf("%d \n", var);
    printf("%d \n", *p_var); //imprime valor NO ENDEREÇO
    printf("%d \n", &var); //imprime endereço de var
    printf("%d \n", p_var); //imprime endereço armazenado

    return 0;
}
```

Operadores

8

```
int main(void)
{
    int var;
    int *p_var; //ponteiro de int

    var = 50;
    p_var = &var; //recebe ENDEREÇO DE var

    printf("%d \n", var);
    printf("%d \n", *p_var); //imprime valor NO ENDEREÇO
    printf("%d \n", &var); //imprime endereço de var
    printf("%d \n", p_var); //imprime endereço armazenado

    return 0;
}
```

"C:\Users\MurIEL\Dropbox\1.

50

50

6356744

6356744

Operadores

9

```
int main(void)
{
    int var;
    int *p_var; //ponteiro de int

    var = 50;
    p_var = &var; //recebe ENDEREÇO DE var

    printf("%d \n", var);
    printf("%d \n", *p_var); //imprime valor NO ENDEREÇO
    printf("%d \n", &var); //imprime endereço de var
    printf("%d \n", p_var); //imprime endereço armazenado

    return 0;
}
```

	"C:\Users\MurIEL\Dropbox\1.
var	50
*p_var	50
&var	6356744
p_var	6356744

Passagem de parâmetros

10

```
#include <stdio.h>

void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}

int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;

    troca(&X, &Y);

    printf("X = %d\n", X);
    printf("Y = %d\n", Y);

    return 0;
}
```

MEMÓRIA

Passagem de parâmetros

11

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    → int X, Y;

    X = 5;
    Y = 10;

    troca(&X, &Y);

    printf("X = %d\n", X);
    printf("Y = %d\n", Y);

    return 0;
}
```

MEMÓRIA

0x002	X	
0x005	Y	

Passagem de parâmetros

12

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    → X = 5;
    Y = 10;

    troca(&X, &Y);

    printf("X = %d\n", X);
    printf("Y = %d\n", Y);

    return 0;
}
```

MEMÓRIA

0x002	X	5
0x005	Y	

Passagem de parâmetros

13

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    → Y = 10;

    troca(&X, &Y);

    printf("X = %d\n", X);
    printf("Y = %d\n", Y);

    return 0;
}
```

MEMÓRIA

0x002	X	5
0x005	Y	10

Passagem de parâmetros

14

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;

    → troca(&X, &Y);

    printf("X = %d\n", X);
    printf("Y = %d\n", Y);

    return 0;
}
```

MEMÓRIA

0x002	X	5
0x005	Y	10

Passagem de parâmetros

15

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;
```

II → `troca(&X, &Y);`

```
printf("X = %d\n", X);
printf("Y = %d\n", Y);

return 0;
}
```

MEMÓRIA

0x002	X	5
0x005	Y	10

Função
chamadora entra
em espera

Passagem de parâmetros

```
16 #include <stdio.h>

→ void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}

int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;

    II → troca(&X, &Y);

    printf("X = %d\n", X);
    printf("Y = %d\n", Y);

    return 0;
}
```

Inicia função
declarando
parâmetros...

MEMÓRIA

0x002	X	5
0x005	Y	10

0x029	A	
0x012	B	

Passagem de parâmetros

17 `#include <stdio.h>`

→ `void troca(int* A, int* B)`

```
{  
    int aux;  
  
    aux = *A;  
    *A = *B;  
    *B = aux;  
}
```

```
int main(void)  
{
```

```
    int X, Y;
```

```
    X = 5;  
    Y = 10;
```

... e inicializa com valores recebidos

II → `troca(&X, &Y);`

```
    printf("X = %d\n", X);  
    printf("Y = %d\n", Y);
```

```
    return 0;
```

```
}
```

MEMÓRIA

0x002	X	5
0x005	Y	10

0x029	A	0x002
0x012	B	0x005

Passagem de parâmetros

18

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

```
void troca(int* A, int* B)
{
    → int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;
```

II → `troca(&X, &Y);`

```
printf("X = %d\n", X);
printf("Y = %d\n", Y);

return 0;
}
```

MEMÓRIA

0x002	X	5
0x005	Y	10

0x035	aux	
-------	-----	--

0x029	A	0x002
0x012	B	0x005

Passagem de parâmetros

19

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

```
void troca(int* A, int* B)
{
    int aux;

    → aux = *A;
      *A = *B;
      *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;
```

II → `troca(&X, &Y);`

```
printf("X = %d\n", X);
printf("Y = %d\n", Y);

return 0;
}
```

MEMÓRIA

0x002	X	5
0x005	Y	10

0x035	aux	
-------	-----	--

0x029	A	0x002
0x012	B	0x005

Passagem de parâmetros

20

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

```
void troca(int* A, int* B)
{
    int aux;

    → aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;
```

```
II → troca(&X, &Y);
```

```
printf("X = %d\n", X);
printf("Y = %d\n", Y);

return 0;
```

MEMÓRIA

0x002	X	5
0x005	Y	10

0x035	aux	5
-------	-----	---

0x029	A	0x002
0x012	B	0x005

Acessado o **valor**
dentro do
endereço que A
armazena

Passagem de parâmetros

21

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    → *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;
```

II → `troca(&X, &Y);`

```
printf("X = %d\n", X);
printf("Y = %d\n", Y);

return 0;
```

MEMÓRIA

0x002	X	5
0x005	Y	10

0x035	aux	5
-------	-----	---

0x029	A	0x002
0x012	B	0x005

Passagem de parâmetros

22

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    → *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;

    II → troca(&X, &Y);

    printf("X = %d\n", X);
    printf("Y = %d\n", Y);

    return 0;
}
```

Acessando os
valores dentro
do endereço que
A e B armazenam

MEMÓRIA

0x002	X	5
0x005	Y	10

0x035	aux	5
-------	-----	---

0x029	A	0x002
0x012	B	0x005

Passagem de parâmetros

23

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    → *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;
```

II → `troca(&X, &Y);`

```
printf("X = %d\n", X);
printf("Y = %d\n", Y);

return 0;
}
```

MEMÓRIA

0x002	X	10
0x005	Y	10

0x035	aux	5
-------	-----	---

0x029	A	0x002
0x012	B	0x005

Passagem de parâmetros

24

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    → *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;
```

II → `troca(&X, &Y);`

```
printf("X = %d\n", X);
printf("Y = %d\n", Y);

return 0;
}
```

MEMÓRIA

0x002	X	10
0x005	Y	5

0x035	aux	5
-------	-----	---

0x029	A	0x002
0x012	B	0x005

Passagem de parâmetros

25

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;

```

```
    troca(&X, &Y);
```

```
    printf("X = %d\n", X);
    printf("Y = %d\n", Y);
```

```
    return 0;
}
```

Fim da execução
da função.
**Variáveis locais
são desalocadas.**

MEMÓRIA

0x002	X	10
0x005	Y	5

0x035	aux	5
-------	-----	---

0x029	A	0x002
0x012	B	0x005

Passagem de parâmetros

26

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;
```

```
    X = 5;
    Y = 10;
```

```
    → troca(&X, &Y);
```

```
    printf("X = %d\n", X);
    printf("Y = %d\n", Y);
```

```
    return 0;
```

```
}
```

MEMÓRIA

0x002	X	10
0x005	Y	5

Função
chamadora volta
à execução

Passagem de parâmetros

27

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;

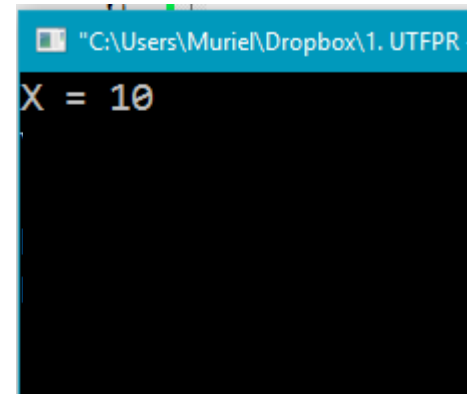
    troca(&X, &Y);

    → printf("X = %d\n", X);
    printf("Y = %d\n", Y);

    return 0;
}
```

MEMÓRIA

0x002	X	10
0x005	Y	5



```
"C:\Users\Muriele\Dropbox\1. UTFPR -
X = 10
Y = 5
```

Passagem de parâmetros

28

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;

    troca(&X, &Y);

    printf("X = %d\n", X);
    → printf("Y = %d\n", Y);

    return 0;
}
```

MEMÓRIA

0x002	X	10
0x005	Y	5

```
"C:\Users\MurIEL\Dropbox\1. UTFPR -
X = 10
Y = 5

Process returned 0 (0
Press any key to cont
```

Passagem de parâmetros

29

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

```
void troca(int* A, int* B)
{
    int aux;

    aux = *A;
    *A = *B;
    *B = aux;
}
```

```
int main(void)
{
    int X, Y;

    X = 5;
    Y = 10;

    troca(&X, &Y);

    printf("X = %d\n", X);
    printf("Y = %d\n", Y);

    → return 0;
}
```

MEMÓRIA

0x002	X	10
0x005	Y	5

Fim da execução
do código

```
"C:\Users\MurIEL\Dropbox\1. UTFPR -
X = 10
Y = 5

Process returned 0 (0
Press any key to cont
```

Aritmética de ponteiros

30

- Deve-se tomar o cuidado que a adição e a subtração dependem do tamanho do tipo do dado.

```
char var_char = '0';  
int var_int = 0;
```

```
char *p_char;  
int *p_int;
```

```
p_char = &var_char;  
p_int = &var_int;
```

```
p_char++; //desloca 1 byte  
p_int++; //desloca 4 bytes
```

Operadores relacionais

31

- A comparação entre ponteiros também é válida, seguindo os mesmos princípios dos dados numéricos.

```
int main(void)
{
    char A, B;
    char *p1, *p2;

    p1 = &A;
    p2 = &B;

    //==, !=, >, >=, <, <=
    if(p1 == p2)
        printf("Mesmo endereco.\n");
    else
        printf("Enderecos diferentes.\n");

    return 0;
}
```

Vetores

32

- Vetores são referenciados por um nome e pelos índices.
 - O nome de um vetor é o ponteiro para o primeiro índice.
 - Os índices informam a quantidade de deslocamentos a partir da posição zero.

Vetores

33

```
int main(void)
{
    int vetor[5];
    vetor[0] = 5;
    vetor[1] = 10;
    vetor[2] = 15;
    vetor[3] = 20;
    vetor[4] = 25;

    printf("Vet[%d] = %d \n", 0, vetor[0]);
    printf("Vet[%d] = %d \n", 1, vetor[1]);
    printf("Vet[%d] = %d \n", 2, vetor[2]);
    printf("Vet[%d] = %d \n", 3, vetor[3]);
    printf("Vet[%d] = %d \n", 4, vetor[4]);

    return 0;
}
```

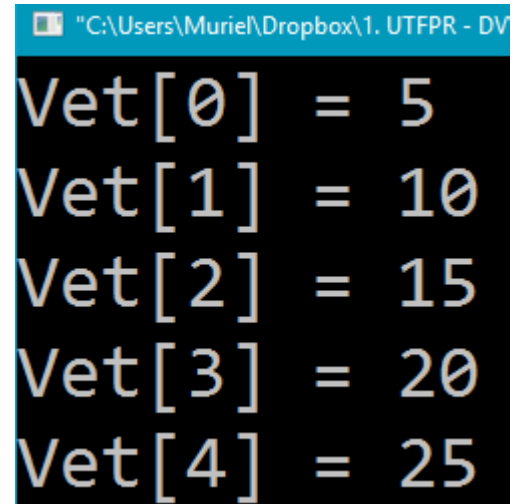
Vetores

34

```
int main(void)
{
    int vetor[5];
    vetor[0] = 5;
    vetor[1] = 10;
    vetor[2] = 15;
    vetor[3] = 20;
    vetor[4] = 25;

    printf("Vet[%d] = %d \n", 0, vetor[0]);
    printf("Vet[%d] = %d \n", 1, vetor[1]);
    printf("Vet[%d] = %d \n", 2, vetor[2]);
    printf("Vet[%d] = %d \n", 3, vetor[3]);
    printf("Vet[%d] = %d \n", 4, vetor[4]);

    return 0;
}
```



```
"C:\Users\MurIEL\Dropbox\1. UTFPR - DV"
Vet[0] = 5
Vet[1] = 10
Vet[2] = 15
Vet[3] = 20
Vet[4] = 25
```

Vetores

35

```
int main(void)
{
    int vetor[5];
    *vetor = 5;
    *(vetor + 1) = 10;
    *(vetor + 2) = 15;
    *(vetor + 3) = 20;
    *(vetor + 4) = 25;

    printf("Vet[%d] = %d \n", 0, *(vetor + 0));
    printf("Vet[%d] = %d \n", 1, *(vetor + 1));
    printf("Vet[%d] = %d \n", 2, *(vetor + 2));
    printf("Vet[%d] = %d \n", 3, *(vetor + 3));
    printf("Vet[%d] = %d \n", 4, *(vetor + 4));

    return 0;
}
```

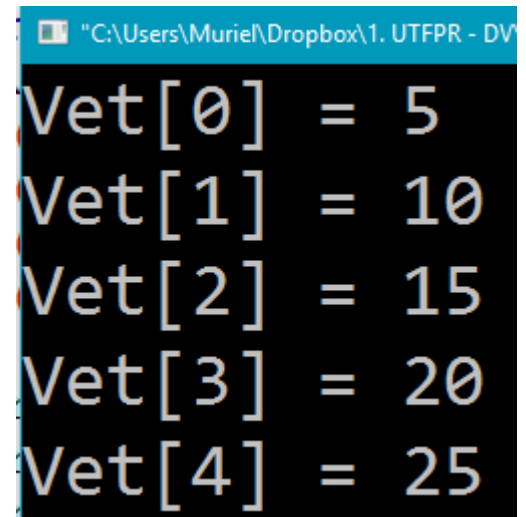
Vetores

36

```
int main(void)
{
    int vetor[5];
    *vetor = 5;
    *(vetor + 1) = 10;
    *(vetor + 2) = 15;
    *(vetor + 3) = 20;
    *(vetor + 4) = 25;

    printf("Vet[%d] = %d \n", 0, *(vetor + 0));
    printf("Vet[%d] = %d \n", 1, *(vetor + 1));
    printf("Vet[%d] = %d \n", 2, *(vetor + 2));
    printf("Vet[%d] = %d \n", 3, *(vetor + 3));
    printf("Vet[%d] = %d \n", 4, *(vetor + 4));

    return 0;
}
```



```
"C:\Users\MurIEL\Dropbox\1. UTFPR - DV"
Vet[0] = 5
Vet[1] = 10
Vet[2] = 15
Vet[3] = 20
Vet[4] = 25
```

Vetores

37

```
int main(void)
{
    int vetor[5];
    *vetor = 5;
    *(vetor + 1) = 10;
    *(vetor + 2) = 15;
    *(vetor + 3) = 20;
    *(vetor + 4) = 25;
```

```
printf("Vet[%d] = %d \n", 0, *(vetor + 0));
printf("Vet[%d] = %d \n", 1, *(vetor + 1));
printf("Vet[%d] = %d \n", 2, *(vetor + 2));
printf("Vet[%d] = %d \n", 3, *(vetor + 3));
printf("Vet[%d] = %d \n", 4, *(vetor + 4));
```

```
return 0;
```

```
}
```

Operações aritméticas do ponteiro respeitam o tamanho do tipo de dado.

Ponteiros genéricos

38

- Pode apontar para todos os tipos de dados existentes.
- Ponteiro do tipo ***void****.

Ponteiros genéricos

39

- Pode apontar para todos os tipos de dados existentes.
- Ponteiro do tipo **void***.

```
int main(void)
{
    char var_c = 'A';
    char *pont_c;

    void *ponteiro;//ponteiro genérico

    ponteiro = &var_c;
    printf("Char: %p\n", &var_c);
    printf("Ponteiro: %p\n", ponteiro);

    ponteiro = &pont_c;//endereço de ponteiro
    printf("Char*: %p\n", &pont_c);
    printf("Ponteiro: %p\n", ponteiro);

    return 0;
}
```

Ponteiros genéricos

40

- Para acessar ao valor do endereço genérico, deve ser utilizado um modificador de tipo (cast).

Ponteiros genéricos

41

- Para acessar ao valor do endereço genérico, deve ser utilizado um modificador de tipo (cast).

```
int main(void)
{
    char var_c = 'A';
    char *pont_c;

    void *ponteiro;//ponteiro genérico

    ponteiro = &var_c;
    printf("Char: %c\n", var_c);
    printf("Ponteiro: %c\n", *(char*)ponteiro);

    return 0;
}
```

Ponteiros genéricos

42

- Para acessar ao valor do endereço utilizado um modificador de

Conversão para o **tipo de ponteiro** da operação.

```
int main(void)
{
    char var_c = 'A';
    char *pont_c;

    void *ponteiro;//ponteiro genérico

    ponteiro = &var_c;
    printf("Char: %c\n", var_c);
    printf("Ponteiro: %c\n", *((char*)ponteiro));

    return 0;
}
```

Ponteiros genéricos

43

- A **falta de definição do tipo** do ponteiro genérico gera a necessidade de cuidados:
 - ▣ Para acessar valores sempre deve se converter para o tipo de ponteiro utilizado (cast).
 - ▣ As operações aritméticas sempre utilizam 1 byte.

Ponteiros

44

□ Vantagens:

- ▣ Maior liberdade na manipulação de variáveis e no uso de funções;
- ▣ Uso de alocação dinâmica de memória.

□ Desvantagens:

- ▣ Possibilidade de acessar posições inexistentes ou não alocadas.
 - Inicializar ponteiros vazios com NULL.

Questionário

45

- 1) O que é um ponteiro?
- 2) Explique a relação existente entre ponteiros e vetores.
- 3) Quais as vantagens e desvantagens do uso de ponteiros?

Questionário

46

□ 4) No código a seguir, o que será impresso em tela?

```
int main(void)
{
    int x[3];
    int *A, *B;

    *x = 380;
    *(x + 1) = 2;
    *(x + x[1]) = 45;
    A = &x;
    B = &x[2];

    printf("X[0] = %d \n", x[0]);
    printf("X[1] = %d \n", *(x + 1));
    printf("X[2] = %d \n", x[2]);

    printf("1: %d \n", *A);
    printf("2: %d \n", *B);
    printf("3: %d \n", A);
    printf("4: %d \n", B);

    return 0;
}
```

Questionário

47

- 4) No código a seguir, o que será impresso em tela?

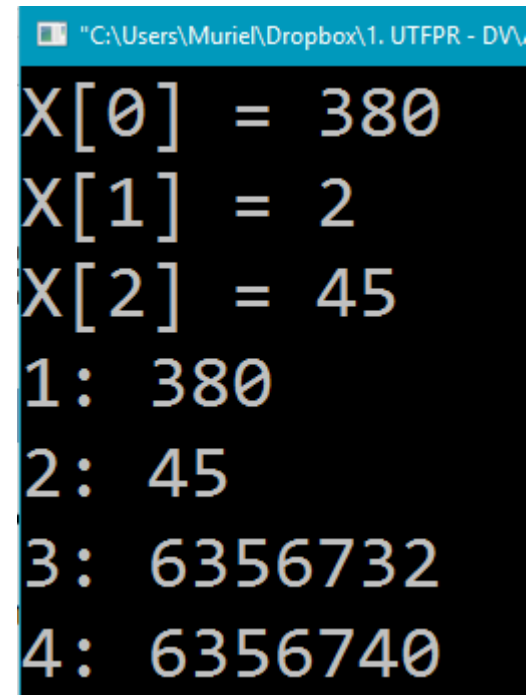
```
int main(void)
{
    int x[3];
    int *A, *B;

    *x = 380;
    *(x + 1) = 2;
    *(x + x[1]) = 45;
    A = &x;
    B = &x[2];

    printf("X[0] = %d \n", x[0]);
    printf("X[1] = %d \n", *(x + 1));
    printf("X[2] = %d \n", x[2]);

    printf("1: %d \n", *A);
    printf("2: %d \n", *B);
    printf("3: %d \n", A);
    printf("4: %d \n", B);

    return 0;
}
```



```
"C:\Users\Muriel\Dropbox\1. UTFPR - DV\
X[0] = 380
X[1] = 2
X[2] = 45
1: 380
2: 45
3: 6356732
4: 6356740
```

Ponteiro de função

48

- O nome de uma função indica o endereço de memória em que ela foi declarada.
- É possível criar ponteiros que armazenam o endereço de funções.
 - ▣ Ponteiro de função.
- Maior utilidade em conjunto das *structs*.

Ponteiro de função

49

```
void maior(int a, int b)
{
    if(a > b) printf("%d maior que %d.\n\n", a, b);
    else printf("%d maior que %d.\n\n", b, a);
}

void menor(int a, int b)
{
    if(a < b) printf("%d menor que %d.\n\n", a, b);
    else printf("%d menor que %d.\n\n", b, a);
}
```

Ponteiro de função

50

```
int main(void)
{
    //ponteiro com mesmo protótipo das funções que irá armazenar
    void (*funcao) (int x, int y);

    int x = 10, y = 200;

    //atribuir o nome da função equivale a passar o endereço da função original
    funcao = maior;
    //utilizar igual uma função qualquer
    funcao(x, y);

    funcao = menor;
    funcao(x, y);

    return 0;
}
```

Ponteiro de função

51

```
int main(void)
{
    //ponteiro com
    void (*funcao)

    int x = 10, y

    //atribuir o n
    funcao = maior
    //utilizar igu
    funcao(x, y);

    funcao = menor;
    funcao(x, y);

    return 0;
}
```

200 maior que 10.

10 menor que 200.

nar

da função original

Vetor de ponteiro de função

52

- É possível criar vetor com esses ponteiros.
 - ▣ Obrigatoriamente só armazenará funções de mesmo protótipo.

Vetor de ponteiro de função

53

```
void funcao_A(void)
{
    printf("Estou na funcao A.\n");
}

void funcao_B(void)
{
    printf("Estou na funcao B.\n");
}

void funcao_C(void)
{
    printf("Estou na funcao C.\n");
}

void funcao_D(void)
{
    printf("Estou na funcao D.\n");
}
```

Vetor de ponteiro de função

54

```
void funcao_A(void)
{
    printf("Estou na funcao A.\n");
}

void funcao_B(void)
{
    printf("Estou na funcao B.\n");
}

void funcao_C(void)
{
    printf("Estou na funcao C.\n");
}

void funcao_D(void)
{
    printf("Estou na funcao D.\n");
}
```

```
int main(void)
{
    void (*funcoes[4]) (void);

    funcoes[0] = funcao_A;
    funcoes[1] = funcao_B;
    funcoes[2] = funcao_C;
    funcoes[3] = funcao_D;

    int i;

    for(i = 0; i < 4; i++)
    {
        funcoes[i]();
    }

    return 0;
}
```

Vetor de ponteiro de função

55

```
void funcao_A(void)
{
    printf("Estou na funcao A.\n");
}

void funcao_B(void)
{
    printf("Estou na funcao B.\n");
}

void funcao_C(void)
{
    printf("Estou na funcao C.\n");
}

void funcao_D(void)
{
    printf("Estou na funcao D.\n");
}

int main(void)
{
    void (*funcoes[4]) (void);
    funcoes[0] = funcao_A;
    funcoes[1] = funcao_B;
    funcoes[2] = funcao_C;
    funcoes[3] = funcao_D;

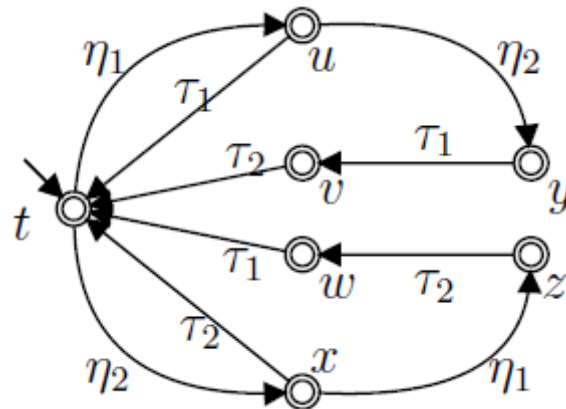
    for (i = 0; i < 4; i++)
    {
        funcoes[i] ();
    }

    return 0;
}
```

Vetor de ponteiro de função

56

- Os vetores de função são úteis para implementar máquinas de estado.
 - ▣ Grafos, Autômatos, Redes de Markov, etc.



Vetor de ponteiro de função

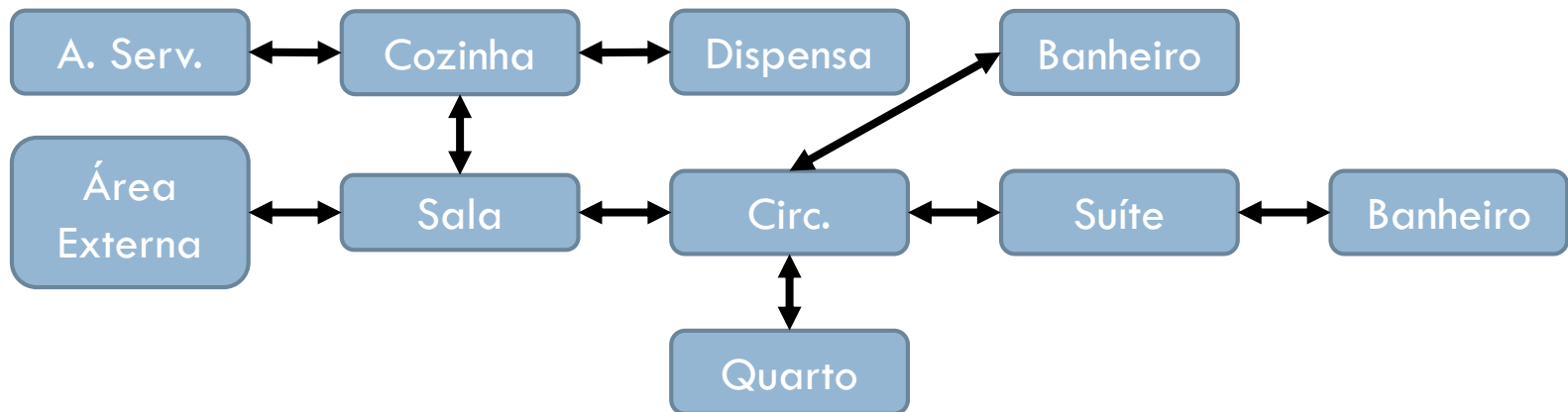
57

- Cada cômodo de uma casa pode ser visto como um estado.
- A transição entre esses estados é feita através das portas.



Vetor de ponteiro de função

58



Vetor de ponteiro de função

59

- Abra o exemplo de ponteiros disponível no Moodle.
- Execute e complete o código de vetor de ponteiro de função.