Faculdade de Computação - FACOM

Bacharelado em Sistemas de Informação

FACOM32201 - Algoritmos e Programação II

Prof. Thiago Pirola Ribeiro

Ponteiros - continuação

Operações com ponteiros

- Atribuição:
  - `p1` aponta para o mesmo lugar que `p2`;

    ```
    p1 = p2;
    ```

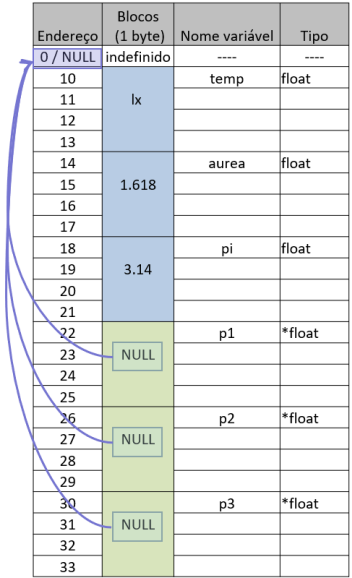
  - a variável apontada por `p1` recebe o mesmo conteúdo da variável apontada por `p2`;

    ```
    *p1 = *p2;
    ```

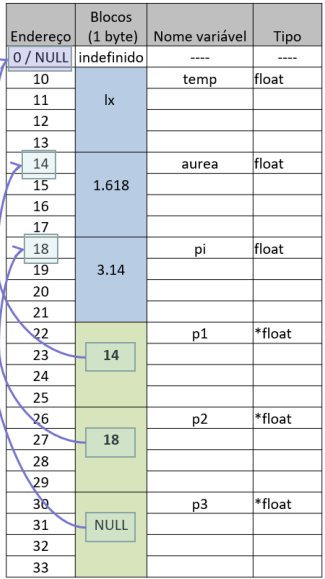
```
1  float temp, aurea = 1.618, pi = 3.14;
2  float *p1 = NULL;
3  float *p2 = NULL;
4  float *p3 = NULL;
5  --> Parou Aqui <--
6  // atribuindo os ponteiros
7  p1 = &aurea;
8  p2 = &pi;
9
10 // copiando os ponteiros
11 p3 = p1;
12 p1 = p2;
13 p2 = p3;
14
15 // atualizando p3
16 p3 = &temp;
17
18 // operação matemática via ponteiro
19 *p3 = *p1 - *p2
20
21 // mesma operação matemática sem uso dos ponteiro
22 temp = pi - aurea;
```

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
10	lx	temp	float
11			
12			
13			
14	1.618	aurea	float
15			
16			
17			
18	3.14	pi	float
19			
20			
21			
22	NULL	p1	*float
23			
24			
25			
26	NULL	p2	*float
27			
28			
29			
30	NULL	p3	*float
31			
32			
33			

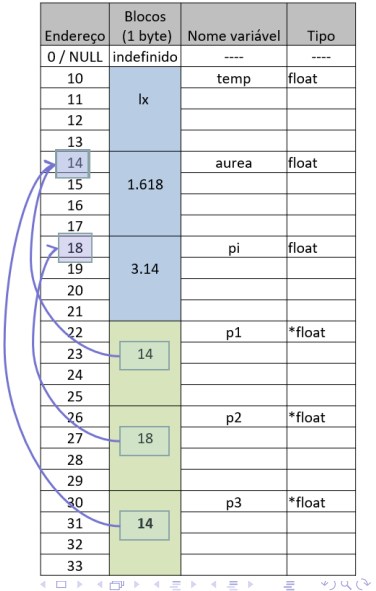
```c
float temp, aurea = 1.618, pi = 3.14;
float *p1 = NULL;
float *p2 = NULL;
float *p3 = NULL;
--> Parou Aqui <--
// atribuindo os ponteiros
p1 = &aurea;
p2 = &pi;

// copiando os ponteiros
p3 = p1;
p1 = p2;
p2 = p3;

// atualizando p3
p3 = &temp;

// operação matemática via ponteiro
*p3 = *p1 - *p2

// mesma operação matemática sem uso dos ponteiro
temp = pi - aurea;
```



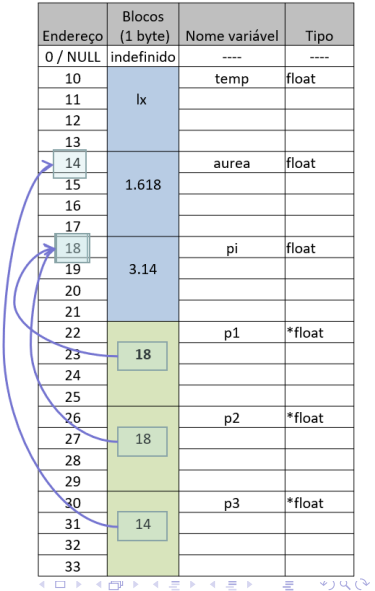
```
1  float temp, aurea = 1.618, pi = 3.14;
2  float *p1 = NULL;
3  float *p2 = NULL;
4  float *p3 = NULL;
5
6  // atribuindo os ponteiros
7  p1 = &aurea;
8  p2 = &pi;
9  --> Parou Aqui <--
10 // copiando os ponteiros
11 p3 = p1;
12 p1 = p2;
13 p2 = p3;
14
15 // atualizando p3
16 p3 = &temp;
17
18 // operação matemática via ponteiro
19 *p3 = *p1 - *p2
20
21 // mesma operação matemática sem uso dos ponteiro
22 temp = pi - aurea;
```



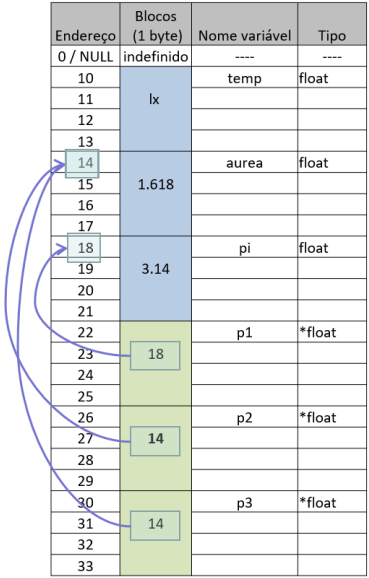
```
1  float temp, aurea = 1.618, pi = 3.14;
2  float *p1 = NULL;
3  float *p2 = NULL;
4  float *p3 = NULL;
5
6  // atribuindo os ponteiros
7  p1 = &aurea;
8  p2 = &pi;
9
10 // copiando os ponteiros
11 p3 = p1;
12 --> Parou Aqui <--
13 p1 = p2;
14 p2 = p3;
15
16 // atualizando p3
17 p3 = &temp;
18
19 // operação matemática via ponteiro
20 *p3 = *p1 - *p2
21
22 // mesma operação matemática sem uso dos ponteiro
23 temp = pi - aurea;
```



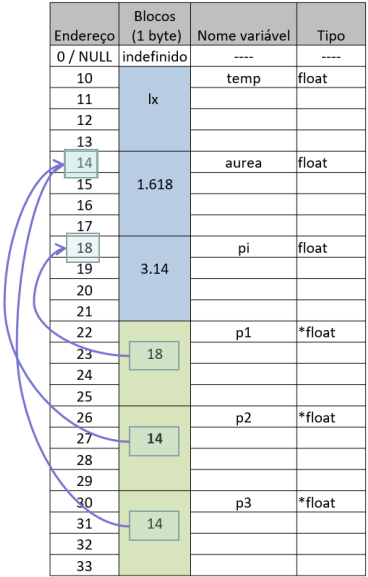
```
1  float temp, aurea = 1.618, pi = 3.14;
2  float *p1 = NULL;
3  float *p2 = NULL;
4  float *p3 = NULL;
5
6  // atribuindo os ponteiros
7  p1 = &aurea;
8  p2 = &pi;
9
10 // copiando os ponteiros
11 p3 = p1;
12 p1 = p2;
13 --> Parou Aqui <--
14 p2 = p3;
15
16 // atualizando p3
17 p3 = &temp;
18
19 // operação matemática via ponteiro
20 *p3 = *p1 - *p2
21
22 // mesma operação matemática sem uso dos ponteiro
23 temp = pi - aurea;
```



```
1  float temp, aurea = 1.618, pi = 3.14;
2  float *p1 = NULL;
3  float *p2 = NULL;
4  float *p3 = NULL;
5
6  // atribuindo os ponteiros
7  p1 = &aurea;
8  p2 = &pi;
9
10 // copiando os ponteiros
11 p3 = p1;
12 p1 = p2;
13 p2 = p3;
14 --> Parou Aqui <--
15 // atualizando p3
16 p3 = &temp;
17
18 // operação matemática via ponteiro
19 *p3 = *p1 - *p2
20
21 // mesma operação matemática sem uso dos ponteiro
22 temp = pi - aurea;
```



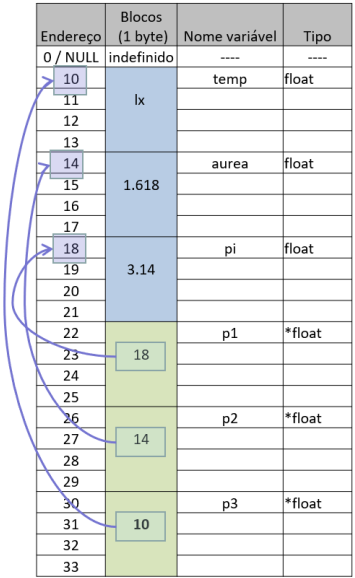
```c
float temp, aurea = 1.618, pi = 3.14;
float *p1 = NULL;
float *p2 = NULL;
float *p3 = NULL;

// atribuindo os ponteiros
p1 = &aurea;
p2 = &pi;

// copiando os ponteiros
p3 = p1;
p1 = p2;
p2 = p3;

--> Parou Aqui <--
// atualizando p3
p3 = &temp;

// operação matemática via ponteiro
*p3 = *p1 - *p2

// mesma operação matemática sem uso dos ponteiro
temp = pi - aurea;
```



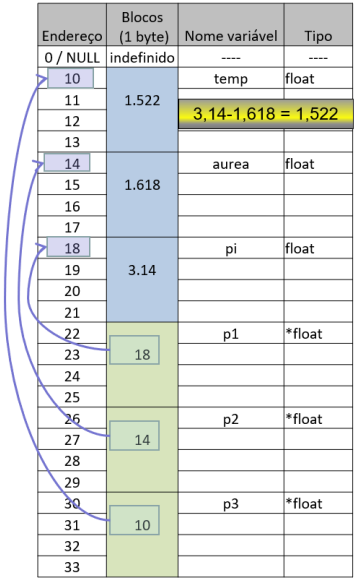
Observação

Foi realizada a operação de **TROCA** entre os ponteiros p1 e p2.

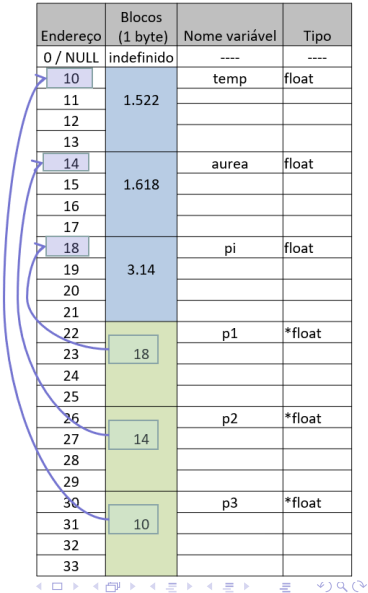
```
1  float temp, aurea = 1.618, pi = 3.14;
2  float *p1 = NULL;
3  float *p2 = NULL;
4  float *p3 = NULL;
5
6  // atribuindo os ponteiros
7  p1 = &aurea;
8  p2 = &pi;
9
10 // copiando os ponteiros
11 p3 = p1;
12 p1 = p2;
13 p2 = p3;
14
15 // atualizando p3
16 p3 = &temp;
17 --> Parou Aqui <--
18 // operação matemática via ponteiro
19 *p3 = *p1 - *p2
20
21 // mesma operação matemática sem uso dos ponteiro
22 temp = pi - aurea;
```



```
1  float temp, aurea = 1.618, pi = 3.14;
2  float *p1 = NULL;
3  float *p2 = NULL;
4  float *p3 = NULL;
5
6  // atribuindo os ponteiros
7  p1 = &aurea;
8  p2 = &pi;
9
10 // copiando os ponteiros
11 p3 = p1;
12 p1 = p2;
13 p2 = p3;
14
15 // atualizando p3
16 p3 = &temp;
17
18 // operação matemática via ponteiro
19 *p3 = *p1 - *p2
20 --> Parou Aqui <--
21 // mesma operação matemática sem uso dos ponteiro
22 temp = pi - aurea;
```



```
1  float temp, aurea = 1.618, pi = 3.14;
2  float *p1 = NULL;
3  float *p2 = NULL;
4  float *p3 = NULL;
5
6  // atribuindo os ponteiros
7  p1 = &aurea;
8  p2 = &pi;
9
10 // copiando os ponteiros
11 p3 = p1;
12 p1 = p2;
13 p2 = p3;
14
15 // atualizando p3
16 p3 = &temp;
17
18 // operação matemática via ponteiro
19 *p3 = *p1 - *p2
20
21 // mesma operação matemática sem uso dos ponteiro
22 temp = pi - aurea;
23 --> Parou Aqui <--
```



Operações com ponteiros

- Apenas duas operações aritméticas podem ser utilizadas com o endereço armazenado pelo ponteiro: **adição** e **subtração**

- Pode-se apenas somar e subtrair valores INTEIROS
  - `p++;` - avança o ponteiro em uma posição;
  - `p--;` - recua o ponteiro em uma posição;
  - `p = p + 15;` - avança 15 posições;
  - `p = p + i;` - avança (`se i>0`) ou retrai (`se i<0`) posições.

Operações com ponteiros

- As operações de adição e subtração no endereço **dependem do tipo de dado** que o ponteiro aponta.
- Considere um ponteiro para inteiro, `int *`. O tipo `int` ocupa um espaço de 4 bytes na memória.
- Assim, nas operações de adição e subtração são adicionados/subtraídos 4 bytes por incremento/decremento, pois esse é o tamanho de um inteiro na memória e, portanto, é também o valor mínimo necessário para sair dessa posição reservada de memória.

```
char *pc = 1;
```

52	1	pc	char *
53			
54			
55			

Ponto 1

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			

```
int *pi = 1;
```

52	1	pi	int *
53			
54			
55			

Ponto 2

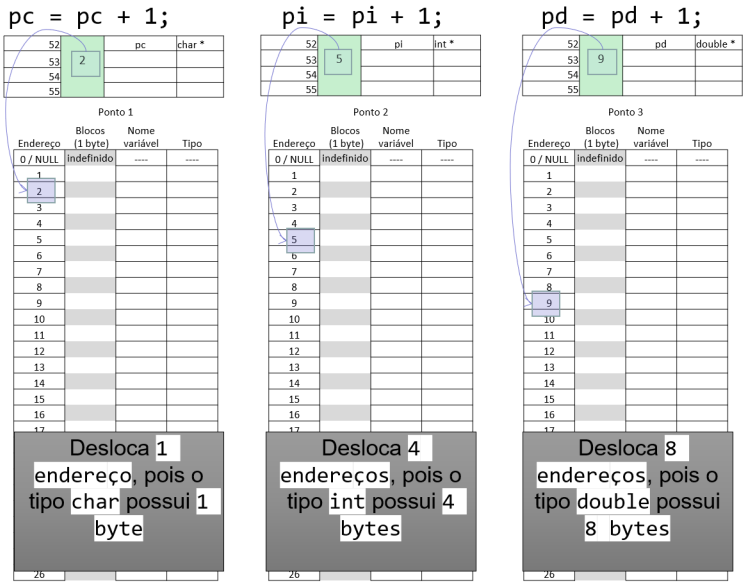
Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			

```
double *pd = 1;
```

52	1	pd	double *
53			
54			
55			

Ponto 3

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			


pc = pc + 1;
52
53
54
55
2
pc
char *
Ponto 1
Endereço
Blocos (1 byte)
Nome variável
Tipo
0 / NULL
indefinido

Desloca 1 endereço, pois o tipo char possui 1 byte
pi = pi + 1;
5
pi
int *
Ponto 2
Desloca 4 endereços, pois o tipo int possui 4 bytes
pd = pd + 1;
9
pd
double *
Ponto 3
Desloca 8 endereços, pois o tipo double possui 8 bytes

```
pc = pc + 1;
```

52	3	pc	char *
53			
54			
55			

Ponto 1

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			

```
pi = pi + 1;
```

52	9	pi	int *
53			
54			
55			

Ponto 2

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			

```
pd = pd + 1;
```

52	17	pd	double *
53			
54			
55			

Ponto 3

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			

Operações com ponteiros

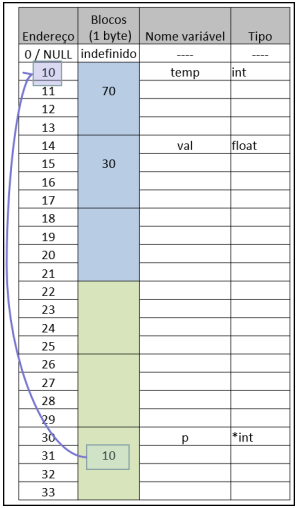
- Operações **ilegais** com ponteiros:
  - Dividir ou multiplicar ponteiros;
  - Somar o endereço de dois ponteiros;
  - Não se pode adicionar ou subtrair `float` ou `double` de ponteiros.

Operações com ponteiros

- Sobre seu conteúdo apontado, valem todas as operações do tipo apontado:
  - `(*p)++;` - incrementar o conteúdo da variável apontada pelo ponteiro p;
  - `*p = (*p) * 15;` - multiplica o conteúdo da variável apontada pelo ponteiro p por 15;
- Devido à precedência dos operadores, é obrigatório ter o parênteses para essa operação
  - Operador pós-fixado

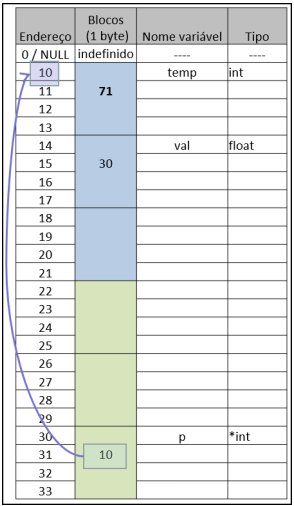
Operações com ponteiros

```
1 --> Parou Aqui <--
2  // alterando temp (forma correta)
3  (*p)++;
4
5
6  // alterando temp (forma incorreta)
7  *p++;
```



Operações com ponteiros

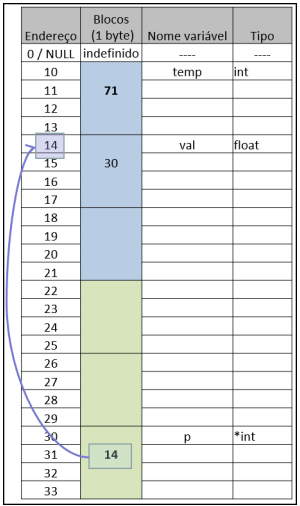
```
1
2  // alterando temp (forma correta)
3  (*p)++;
4  --> Parou Aqui <--
5
6  // alterando temp (forma incorreta)
7  *p++;
```



Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
10		temp	int
11	71		
12			
13			
14		val	float
15	30		
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30		p	*int
31	10		
32			
33			

Operações com ponteiros

```
1
2  // alterando temp (forma correta)
3  (*p)++;
4
5  // alterando temp (forma incorreta)
6  *p++;
7  --> Parou Aqui <--
```



Operações com ponteiros

```
1
2  // alterando temp (forma correta)
3  (*p)++;
4
5  // alterando temp (forma incorreta)
6  *p++;
7  --> Parou Aqui <--
```

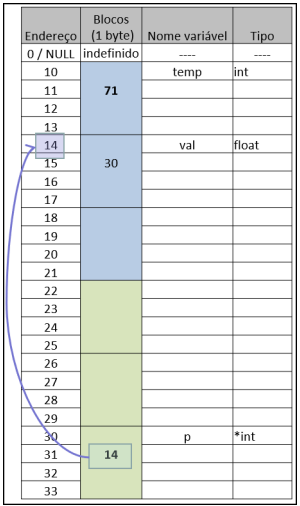
Precedence	Operator	Description
1	`++ --`	Suffix/postfix increment and decrement
	`()`	Function call
	`[]`	Array subscripting
	`.`	Structure and union member access
	`->`	Structure and union member access through pointer
	`(type){list}`	Compound literal(C99)
2	`++ --`	Prefix increment and decrement[note 1]
	`+ -`	Unary plus and minus
	`! ~`	Logical NOT and bitwise NOT
	`(type)`	Cast
	`*`	Indirection (dereference)
	`&`	Address-of
	`sizeof`	Size-of[note 2]
	`_Alignof`	Alignment requirement(C11)

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
10		temp	int
11	71		
12			
13			
14		val	float
15	30		
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30		p	*int
31	14		
32			
33			

Operações com ponteiros

```
1
2  // alterando temp (forma correta)
3  (*p)++;
4
5  // alterando temp (forma incorreta)
6  *p++;
7  --> Parou Aqui <--
```

Executou `p++`, primeiro, mudando o ponteiro para a próxima posição (+4, pode ser do tipo **`int`**) e em seguida fez o deferenciamento (`*p`) de val, que ocorrerá de forma errônea, pois `val` é **`float`**.



Operações com ponteiros

- Operações relacionais
  - == e != para saber se dois ponteiros são iguais ou diferentes.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   int main(){
4      int *p, *p1, x, y;
5      p = &x;
6      p1 = &y;
7      if (p == p1)
8         printf(''Ponteiros iguais\n'');
9      else
10        printf(''Ponteiros diferentes\n'');
11     system(''pause'');
12     returno 0;
13  }
```

Operações com ponteiros

- Operações relacionais
  - >, <, >= e <= para saber qual ponteiro aponta para uma posição mais alta na memória.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4     int *p, *p1, x, y;
5     p = &x;
6     p1 = &y;
7     if (p > p1)
8        printf(''O ponteiro p aponta para uma posição a frente de p1\n'');
9     else
10       printf(''O ponteiro p NAO aponta para uma posição a frente de p1\n'');
11    system(''pause'');
12    returno 0;
13 }
```

Ponteiros Genéricos

- Normalmente, um ponteiro aponta para um tipo específico de dado.
  - Um ponteiro genérico é um ponteiro que pode apontar para qualquer tipo de dado.
  - Operações de soma e subtração (ex. `p++`) deslocam o ponteiro na memória em uma unidade.
- Declaração

```
void *nome_ponteiro;
```

Operações com ponteiros

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

int main()
{
    int a = 10;
    double d = 30;

    void *p;

    // atribuindo o endereço de 'a' ao ponteiro void
    p = &a;

    // mostrando o conteúdo do endereço apontado por p (no caso, a var. 'a')
    printf("Valor de a: %d", *p);
}
```

Operações com ponteiros

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

int main()
{
    int a = 10;
    double d = 30;

    void *p;

    // atribuindo o endereço de 'a' ao ponteiro void
    p = &a;

    // mostrando o conteúdo do endereço apontado por p (no caso, a var. 'a')
    printf("Valor de a: %d", *p);
}
```

- Erro em tempo de compilação:
  - **ERROR: invalid use of void expression.**

Operações com ponteiros

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

int main()
{
    int a = 10;
    double d = 30;

    void *p;

    // atribuindo o endereço de 'a' ao ponteiro void
    p = &a;

    // mostrando o conteúdo do endereço apontado por p (no caso, a var. 'a')
    printf("Valor de a: %d", *(int *)p);
}
```

- Para usar `*void`, é necessário fazer a **conversão para o tipo (typecast)** do ponteiro que o `void` aponta. No caso, é um ponteiro para inteiro.
  - para converter: `(int *)`

Observação sobre o cast

- Esse código só funcionou, pois a linguagem faz conversão (cast) do tipo int para o tipo ponteiro *(Any integer can be cast to any pointer type / Any pointer type can be cast to any integer type*)

```
1  int k;
2  unsigned long int endereco_de_k;
3  // obtendo o endereco da variavel 'k'
4  // será usado o operador &
5  endereco_de_k = &k;
6  scanf("%d",endereco_de_k);
```

Mudando o apontamento de p de um inteiro para um double

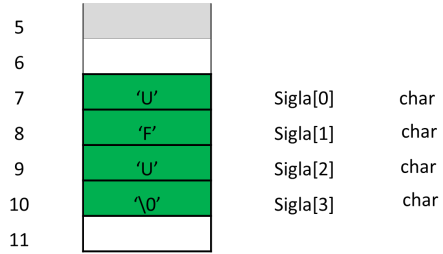
```
1     int a = 10;
2     double d = 30;
3
4     void *p;
5
6     // atribuindo o endereço de 'a' (inteiro) ao ponteiro void
7     p = &a;
8
9     // mostrando o conteúdo do endereço apontado por p (no caso, a var. 'a')
10 //    printf("Valor de a: %d", *p);      // errado!
11
12    // mostrando o conteúdo do endereço apontado por p (no caso, a var. 'a')
13    printf("Valor de a: %d", *(int *)p);
14
15    // atribuindo o endereço de 'd' (double) ao ponteiro void
16    p = &d;
17
18    // mostrando o conteúdo do endereço apontado por p (no caso, a var. 'd')
19    printf("Valor de b: %f", *(double *)p);
```

Ponteiros e Arrays

- Ponteiros e arrays possuem uma ligação muito forte.
  - Arrays são agrupamentos de dados do mesmo tipo na memória.

Ideia

4 'variáveis' `char` agrupadas.

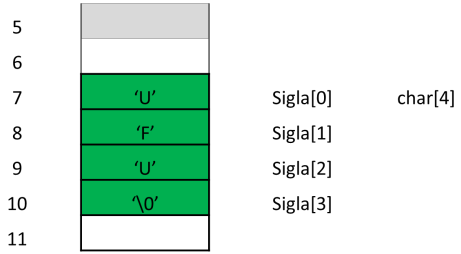


Ponteiros e Arrays

- Ponteiros e arrays possuem uma ligação muito forte.
  - Quando declaramos um array, informamos ao computador para reservar uma certa quantidade de memória a fim de armazenar os elementos do array de forma sequencial.

Ideia

`4 x 1 byte = 4 bytes.`

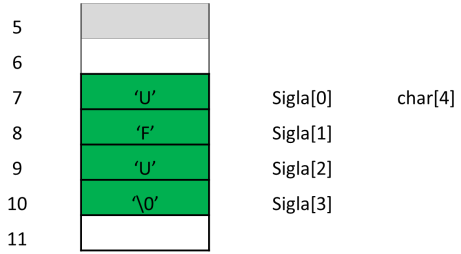


Ponteiros e Arrays

- Ponteiros e arrays possuem uma ligação muito forte.
  - Como resultado dessa operação, o computador nos devolve um ponteiro que aponta para o começo dessa sequência de bytes na memória.

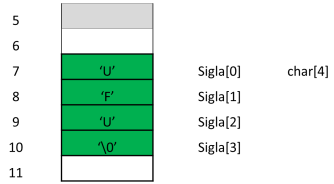
Ideia

Endereço inicial: 7.



Ponteiros e Arrays

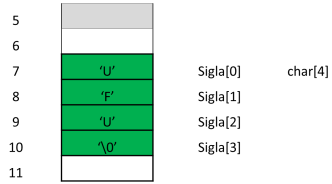
- Em C, o nome do array (sem índice) é apenas um **ponteiro** que aponta para o primeiro elemento do array.




```
1  char Sigla[4] = "UFU";
2
3  // mostrando o endereço da posição 0 do vetor
4   printf("\n Posicao de indice zero do vetor (Sigla[0]): %p",&Sigla[0]);
5
6   // mostrando o endereco da posicao 0 do vetor
7   printf("\n Posicao de indice zero do vetor (Sigla): %p",Sigla);
```

Ponteiros e Arrays

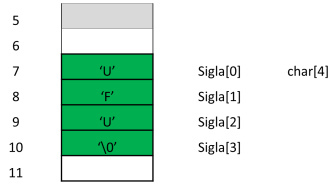
- Em C, o nome do array (sem índice) é apenas **um ponteiro** que aponta para o primeiro elemento do array.
  - **`&Sigla[0]`** é igual **`Sigla`**




```
1  char Sigla[4] = "UFU";
2
3  // mostrando o endereço da posição 0 do vetor
4   printf("\n Posicao de indice zero do vetor (Sigla[0]): %p",&Sigla[0]);
5
6   // mostrando o endereco da posicao 0 do vetor
7   printf("\n Posicao de indice zero do vetor (Sigla): %p",Sigla);
```

Ponteiros e Arrays

- Em C, o nome do array (sem índice) é apenas **um ponteiro** que aponta para o primeiro elemento do array.
  - **&Sigla[0]** é igual **Sigla**




```
1   char Sigla[4] = "UFU";
2
3   // mostrando o endereço da posição 0 do vetor
4   printf("\n Posicao de indice zero do vetor (Sigla[0]): %p",&Sigla[0]);
5
6   // mostrando o endereço da posição 0 do vetor
7   printf("\n Nome do vetor (Sigla): %p",Sigla);
8
9   // mostrando o endereço da posição 0 do vetor
10  printf("\n Endereço do vetor (&Sigla): %p",&Sigla);
```

Ponteiros e Arrays

```
Posicao de indice zero do vetor (Sigla[0]): 000000a6cd5ff73c
Nome do vetor (Sigla): 000000a6cd5ff73c
Endereco do vetor (&Sigla): 000000a6cd5ff73c
Process finished with exit code 46
```

```
1   char Sigla[4] = "UFU";
2
3   // mostrando o endereço da posição 0 do vetor
4   printf("\n Posicao de indice zero do vetor (Sigla[0]): %p",&Sigla[0]);
5
6   // mostrando o endereço da posição 0 do vetor
7   printf("\n Nome do vetor (Sigla): %p",Sigla);
8
9   // mostrando o endereço da posição 0 do vetor
10  printf("\n Endereço do vetor (&Sigla): %p",&Sigla);
```

Ponteiros e Arrays

- Em C, o nome do array (sem índice) é apenas um ponteiro que aponta para o primeiro elemento do array.

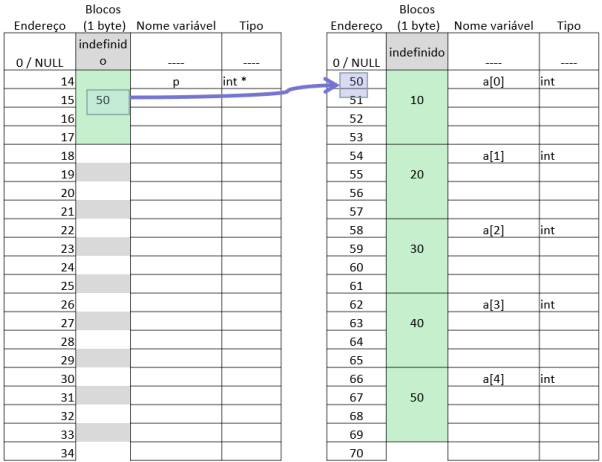
```
1  char Sigla[4] = "UFU";
2  char *p;
3
4  p = Sigla;
```

67			
68			
69	79	p	char *
70			
71			
72			
73			
74			
75			
76			
77			
78			
79	'U'	Sigla[0]	char
80	'F'	Sigla[1]	char
81	'U'	Sigla[2]	char
82	'\0'	Sigla[3]	char
83			
84			

```
int a[5] = {10,20,30,40,50};
int *p;

p = a;
```

--> Parou Aqui <--

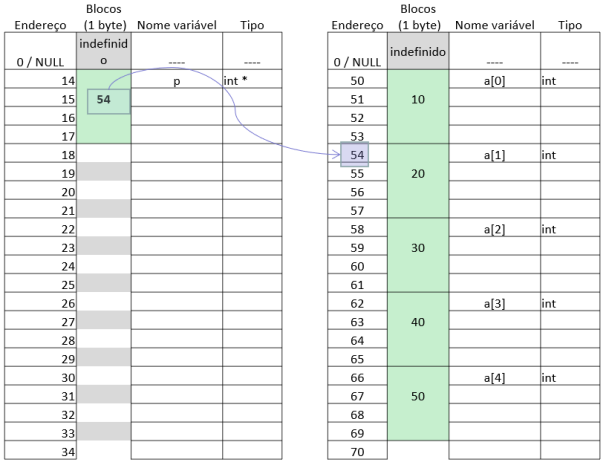
```
p = p + 1;

p = p + 1;

p = p + 2;

p = p + 1;
```



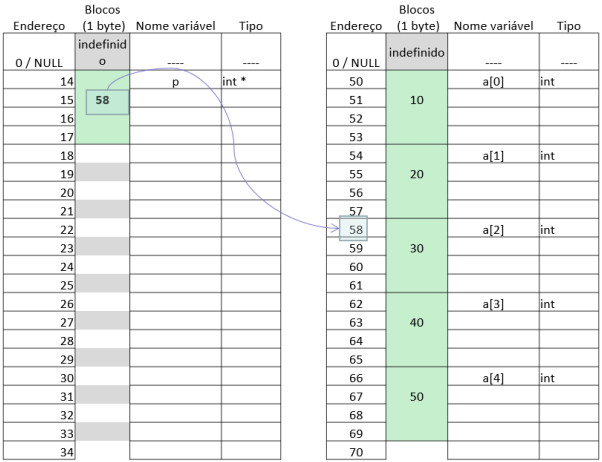
```
int a[5] = {10,20,30,40,50};
int *p;

p = a;

p = p + 1;
```

--> Parou Aqui <--

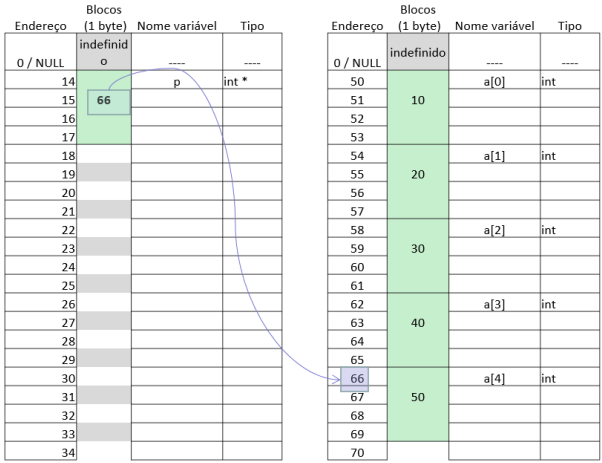
```
p = p + 1;

p = p + 2;

p = p + 1;
```



```
int a[5] = {10,20,30,40,50};
int *p;

p = a;

p = p + 1;

p = p + 1;
--> Parou Aqui <--
p = p + 2;

p = p + 1;
```

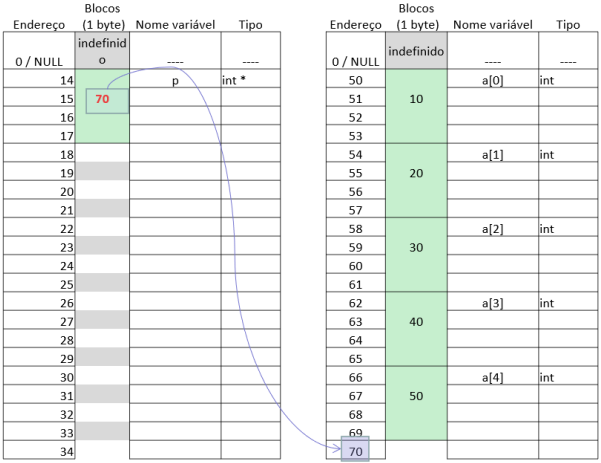


```
int a[5] = {10,20,30,40,50};
int *p;

p = a;

p = p + 1;

p = p + 1;

p = p + 2;
--> Parou Aqui <--
p = p + 1;
```



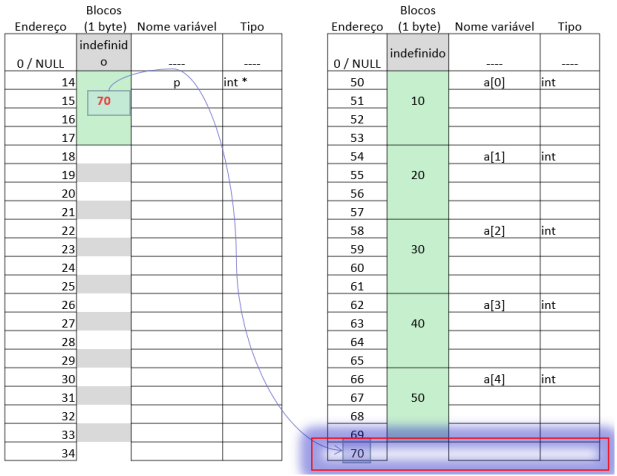
```
int a[5] = {10,20,30,40,50};
int *p;

p = a;

p = p + 1;

p = p + 1;

p = p + 2;

p = p + 1;
```

--> Parou Aqui <--



```
int a[5] = {10,20,30,40,50};
int *p;

p = a;

p = p + 1;

p = p + 1;

p = p + 2;

p = p + 1;
```

--> Parou Aqui <--



Ponteiros e Arrays

- Nesse exemplo:

```
char Sigla[4] = "UFU";
char *p;

p = Sigla;
```

- Tem-se que:
  - **`*p`** é equivalente a **`Sigla[0]`**;
  - **`Sigla[indice]`** é equivalente a **`*(p+indice)`**;
  - **`Sigla`** é equivalente a **`&Sigla[0]`**;
  - **`&Sigla[indice]`** é equivalente a **`(Sigla + indice)`**;

Ponteiros e Arrays

Usando Array

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
    int vet[5] = {1,2,3,4,5};
    int *p = vet;
    for (int i=0; i < 5; i++)
        printf()"%d\n",p[i]);
    return 0;
}
```

Usando Ponteiro

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
    int vet[5] = {1,2,3,4,5};
    int *p = vet;
    for (int i=0; i < 5; i++)
        printf()"%d\n",*(p+i));
    return 0;
}
```

Ponteiros e Arrays

- Os colchetes [ ] substituem o uso conjunto de operações aritméticas e de acesso ao conteúdo (operador $*$) no acesso ao conteúdo de uma posição de um array ou ponteiro.
  - O valor entre colchetes é o deslocamento a partir da posição inicial. Nesse caso, `p[2]` equivale a `*(p+2)`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main (){
4    int vet[5] = {1,2,3,4,5};
5    int *p;
6    p = vet;
7    printf ("Terceiro elemento: %d ou %d",p[2],*(p+2));
8    system("pause");
9    return 0;
10 }
```

Ponteiros e Structs

- Existe um operador específico para trabalhar com deferenciamento de ponteiros para struct
- Operador ->
  - Como usar: `ponteiro -> membro_da_struct`
  - Exemplo

    `(*p).x` é igual a `p->x`

Operators (grouped by precedence)

structure member operator	*name*`.`*member*
structure pointer	*pointer*`->`*member*
increment, decrement	`++ --`
plus, minus, logical not, bitwise not	`+ - ! ~`
indirection via pointer, address of object	`*`*pointer*, `&`*name*
cast expression to type	(*type*) *expr*
size of an object	`sizeof`

```c
struct aluno {
    int num_aluno;
    float nota1, nota2, nota3;
    float media;
};

int main(){

  struct aluno joao;

  joao.num_aluno = 10;
  joao.nota1 = 10;
  joao.nota2 = 4.4;
  joao.nota3 = 7;
  joao.media = (joao.nota1 + joao.nota2 + joao.nota3)/3.0;

  struct aluno *pa;

  pa = &joao;

  printf("Numero aluno: %d\n", (*pa).num_aluno);
  printf("Nota 1: %f\n", (*pa).nota1);
  printf("Nota 2: %f\n", (*pa).nota2);
  printf("Nota 3: %f\n", (*pa).nota3);
  printf("Media 3: %f\n", (*pa).media);
 }
```

```c
struct aluno {
    int num_aluno;
    float nota1, nota2, nota3;
    float media;
};

int main(){

  struct aluno joao;

  joao.num_aluno = 10;
  joao.nota1 = 10;
  joao.nota2 = 4.4;
  joao.nota3 = 7;
  joao.media = (joao.nota1 + joao.nota2 + joao.nota3)/3.0;

```

```c
  struct aluno joao;

  joao.num_aluno = 10;
  joao.nota1 = 10;
  joao.nota2 = 4.4;
  joao.nota3 = 7;
  joao.media = (joao.nota1 + joao.nota2 + joao.nota3)/3.0;

  struct aluno *pa;

  pa = &joao;

  printf("Numero aluno: %d\n", (*pa).num_aluno);
  printf("Nota 1: %f\n", (*pa).nota1);
  printf("Nota 2: %f\n", (*pa).nota2);
  printf("Nota 3: %f\n", (*pa).nota3);
  printf("Media 3: %f\n", (*pa).media);
}
```

```
1   printf("Numero aluno: %d\n", (*pa).num_aluno);
2   printf("Nota 1: %f\n", (*pa).nota1);
3   printf("Nota 2: %f\n", (*pa).nota2);
4   printf("Nota 3: %f\n", (*pa).nota3);
5   printf("Media 3: %f\n", (*pa).media);
6
7   // comandos equivalentes
8   printf("\n");
9   printf("Numero aluno: %d\n", pa->num_aluno);
10  printf("Nota 1: %f\n", pa->nota1);
11  printf("Nota 2: %f\n", pa->nota2);
12  printf("Nota 3: %f\n", pa->nota3);
13  printf("Media 3: %f\n", pa->media);
```

Exercício

- Suponha que os elementos de um `vetor v` são do tipo **`int`** e cada **`int`** ocupa `4 bytes` no computador. Se o endereço de `v[0]` é `55000`, qual o valor da expressão `v + 3`? Justifique.

- Escreva, sem utilizar o computador, quais serão os valores de `x, y e p` ao final do trecho de código:

```
1 int x, y, *p;
2 y = 0;
3 p = &y;
4 x = *p;
5 x = 4;
6 (*p)++;
7 --x;
8 (*p) += x;
```

Exercício

- Crie um programa que contenha uma matriz de `float` contendo 3 linhas e 3 colunas. Imprima o endereço de cada posição dessa matriz.
- Crie um programa que contenha um array de inteiros contendo 5 elementos. Utilizando apenas aritmética de ponteiros, leia esse array do teclado e imprima o dobro de cada valor lido.
- Crie um programa que contenha um array contendo 5 elementos inteiros. Leia esse array do teclado e imprima o endereço das posições contendo valores pares.

Faculdade de Computação - FACOM

Bacharelado em Sistemas de Informação

Prof. Thiago Pirola Ribeiro