



Técnicas de Programação 1

12ª parte

Ponteiros e Alocação Dinâmica de Memória

Prof. Jobson Massollar

jobson@uniriotec.br



Variáveis e Endereços de Memória

Todas as variáveis que declaramos em nossos programas ocupam, em algum momento, algum endereço na memória do computador.

```
int a;  
int b;
```

a: endereço 12

b: endereço 31

	0	1	2	3	4	5	6	7	8	9
0			a							
1			?							
2		b								
3		?								
4										



Variáveis e Endereços de Memória

Quando lemos ou armazenamos dados em uma variável, estamos acessando o endereço de memória associado aquela variável.

```
int a;  
int b;  
  
a = 10;  
b = a;  
a = a + b;
```

Armazena **10** no
endereço **12**.

a: endereço 12

b: endereço 31

	0	1	2	3	4	5	6	7	8	9
0			a							
1			10							
2		b								
3		?								
4										



Variáveis e Endereços de Memória

Quando lemos ou armazenamos dados em uma variável, estamos acessando o endereço de memória associado aquela variável.

```
int a;  
int b;  
  
a = 10;  
b = a;  
a = a + b;
```

Armazena o conteúdo
do endereço **12** no
endereço **31**.

a: endereço 12

b: endereço 31

	0	1	2	3	4	5	6	7	8	9
0			a							
1			10							
2		b								
3		10								
4										



Variáveis e Endereços de Memória

Quando lemos ou armazenamos dados em uma variável, estamos acessando o endereço de memória associado aquela variável.

```
int a;  
int b;  
  
a = 10;  
b = a;  
a = a + b;
```

Soma os conteúdos dos endereços **12** e **31** e armazena o resultado no endereço **12**.

a: endereço 12

b: endereço 31

	0	1	2	3	4	5	6	7	8	9
0			a							
1			20							
2		b								
3		10								
4										



Ponteiros também são variáveis.

Entretanto, ponteiros são variáveis especiais que **não** armazenam dados.

Ponteiros armazenam o **endereço** de outras variáveis.

Por isso dizemos que ponteiros apontam para outras variáveis.



Foto criada por drobotdean - br.freepik.com



Para declarar uma variável ponteiro usamos a seguinte sintaxe:

```
tipo *ptr;
```

Acrescente um * na frente do nome da variável para indicar que ela é um **ponteiro**.



Exemplos:

```
int *p1;
```

p1 é um ponteiro para **int**.

```
char *p2;
```

p2 é um ponteiro para **char**.

```
float *p3;
```

p3 é um ponteiro para **float**.



Note que uma variável ponteiro ocupa um endereço de memória como qualquer outra, afinal uma variável ponteiro também é uma variável!

```
int a;  
int b;  
  
int *p;
```

a: endereço 12

b: endereço 31

p: endereço 26

	0	1	2	3	4	5	6	7	8	9
0			a							
1			?				p			
2		b					?			
3		?								
4										



Como usar a variável `p`? Inicialmente, temos que **inicializá-la**.

```
int a;  
int b;  
  
int *p;
```

a: endereço 12

b: endereço 31

p: endereço 26

	0	1	2	3	4	5	6	7	8	9
0			a							
1			10				p			
2		b					?			
3		?								
4										



O operador **&** colocado na frente de uma variável retorna o **endereço** dessa variável na memória.

```
int a;  
int b;  
  
int *p;  
  
a = 10;  
p = &a;
```

O operador **&** retorna o **endereço** da variável **a** e não seu valor.

a: endereço 12

b: endereço 31

p: endereço 26

	0	1	2	3	4	5	6	7	8	9
0			a							
1			10				p			
2		b					?			
3		?								
4										



O operador `&` colocado na frente de uma variável retorna o **endereço** dessa variável na memória.

```
int a;  
int b;  
  
int *p;  
  
a = 10;  
p = &a;
```

Variáveis ponteiro armazenam o **endereço** de outras variáveis.

Assim, **p** vai receber o **endereço de a** e não o valor de a.

a: endereço 12

b: endereço 31

p: endereço 26

	0	1	2	3	4	5	6	7	8	9
0			a							
1			10				p			
2		b					12			
3		?								
4										



Uma variável ponteiro **aponta** para ou **referencia** outra variável.

```
int a;  
int b;  
  
int *p;  
  
a = 10;  
p = &a;
```

Por essa razão dizemos que:
p aponta para a
ou
p referencia a

a: endereço 12
b: endereço 31
p: endereço 26

	0	1	2	3	4	5	6	7	8	9
0			a							
1			10				p			
2		b					12			
3		?								
4										



O operador `*` recupera o **valor** apontado por um ponteiro.

```
int a;  
int b;  
  
int *p;  
  
a = 10;  
p = &a;  
b = *p;
```

O operador `*` retorna o **valor** apontado pelo ponteiro **p**.

a: endereço 12
b: endereço 31
p: endereço 26

	0	1	2	3	4	5	6	7	8	9
0			a							
1			10				p			
2		b					12			
3		?								
4										



```
int a;  
int b;  
  
int *p;  
  
a = 10;  
p = &a;  
b = *p;
```

a: endereço 12
b: endereço 31
p: endereço 26

[illegible]



O operador `*` recupera o **valor** apontado por um ponteiro.

```
int a;  
int b;  
  
int *p;  
  
a = 10;  
p = &a;  
b = *p;  
*p = *p + b;
```

Soma o **valor** apontado por **p** com a variável **b** e armazena o resultado no endereço de memória apontado por **p**.

a: endereço 12

b: endereço 31

p: endereço 26

	0	1	2	3	4	5	6	7	8	9
0			a							
1			20				p			
2		b					12			
3		10								
4										



O operador `*` recupera o **valor** apontado por um ponteiro.

```
int a;  
int b;  
  
int *p;  
  
a = 10;  
p = &a;  
b = *p;  
*p = *p + b;
```

O efeito final é o mesmo que:
 $a = a + b$

a: endereço 12

b: endereço 31

p: endereço 26

	0	1	2	3	4	5	6	7	8	9
0			a							
1			20				p			
2		b					12			
3		10								
4										



Quando atribuímos **NULL** a um ponteiro estamos dizendo que ele não referencia **nada**.

```
int a;  
int b;  
  
int *p;  
  
a = 10;  
p = &a;  
b = *p;  
*p = *p + b;  
p = NULL;
```

O ponteiro **p** não referencia nenhuma variável.

a: endereço 12

b: endereço 31

p: endereço 26

	0	1	2	3	4	5	6	7	8	9
0			a							
1			20				p			
2		b								
3		10								
4										



Agora, finalmente, entendemos porque a função `scanf` precisa que a variável tenha o `&` na sua frente.

```
int a;  
float b;  
  
scanf("%d", &a);  
scanf("%f", &b);
```

A função **scanf** recebe o **endereço** da variável e não o seu valor.



Importante!

O **&** é usado com dois significados diferentes no C/C++:

- ✓ Quando usado **antes** da declaração de um **parâmetro**, ele indica que esse parâmetro é uma **referência**.

```
void troca(int &x, int &y)
```

- ✓ Quando usado **antes** de uma **variável**, ele retorna o **endereço** dessa variável.

```
int a = 10;  
int *p = &a;
```





12.1) A partir das declarações abaixo, escreva um trecho de código que não usa nem `x` nem `y` e: divide `x` por 5 e armazena o resultado em `y`, soma 10 a `x` e imprime o valor de `y`.

```
int x = 20, y;  
int *p1, *p2;
```



12.1) A partir das declarações abaixo, escreva um trecho de código que não usa nem `x` nem `y` e: divide `x` por 5 e armazena o resultado em `y`, soma 10 a `x` e imprime o valor de `y`.

```
int x = 20, y;  
int *p1, *p2;  
  
p1 = &x;  
p2 = &y;  
*p2 = *p1 / 5;  
*p1 += 10;  
printf("%d\n", *p2);
```



12.2) Qual o resultado das expressões abaixo?

```
int i = 3, j = 5;
```

```
int *p = &i, *q = &j;
```

$*p + 1$

$*p - *q$

$*p * *q$

$(*p + i) - (*q + j)$



12.2) Qual o resultado das expressões abaixo?

```
int i = 3, j = 5;
```

```
int *p = &i, *q = &j;
```

`*p + 1` 4

`*p - *q` -2

`*p * *q` 15

`(*p + i) - (*q + j)` -4



12.3) A chamada à função scanf está correta?

```
int x = 10;  
int *ptr;  
  
ptr = &x;  
scanf("%d", *ptr);
```



12.3) A chamada à função scanf está correta?

```
int x = 10;  
int *ptr;  
  
ptr = &x;  
scanf("%d", *ptr);
```

Não, porque o scanf deve receber o endereço da variável que será lida. Nesse caso, ptr já contém o endereço de x. A chamada correta seria:

```
scanf("%d", ptr);
```



Ponteiros e Operadores Relacionais

Só faz sentido utilizar dois tipos de operadores relacionais com ponteiros:

Operador	Significado
==	Verifica se dois ponteiros referenciam o mesmo endereço de memória.
!=	Verifica se dois ponteiros referenciam endereços de memória diferentes.



Ponteiros e Operadores Relacionais

Exemplo:

```
int a = 10, b = 20;
int *p1 = &a, *p2 = &b;

if (p1 == p2)
    puts("P1 e P2 referenciam o mesmo endereco");

if (p1 != p2)
    puts("P1 e P2 referenciam enderecos distintos");

if (p1 == NULL)
    puts("P1 não referencia nenhum endereco de memoria");

if (p2 != NULL)
    puts("P2 referencia algum endereco de memoria");
```



Aritmética com Ponteiros

Podemos utilizar quatro operadores aritméticos com ponteiros:

Operação	Operador	Significado
Somar um valor inteiro a um ponteiro	+	Faz o ponteiro "avançar" e apontar para uma área de memória posterior à área atual.
Subtrair um ponteiro de um valor inteiro	-	Faz o ponteiro "retroceder" e apontar para uma área de memória anterior à área atual.
Incrementar um ponteiro	++	Faz o ponteiro "avançar" para a área de memória imediatamente posterior.
Decrementar um ponteiro	--	Faz o ponteiro "retroceder" para a área de memória imediatamente anterior.

Essas operações aritméticas são muito usadas quando temos um ponteiro apontando para um **vetor**.



Aritmética com Ponteiros

Exemplo 1:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };  
int *p;
```

Quando criamos um vetor, os elementos ficam **contínuos** na memória.

	0	1	2	3	4	5	6	7	8	9
v	2	4	6	8	10	12	14	16	18	20
	250	251	252	253	254	255	256	257	258	259
p	<div>?</div>									



Aritmética com Ponteiros

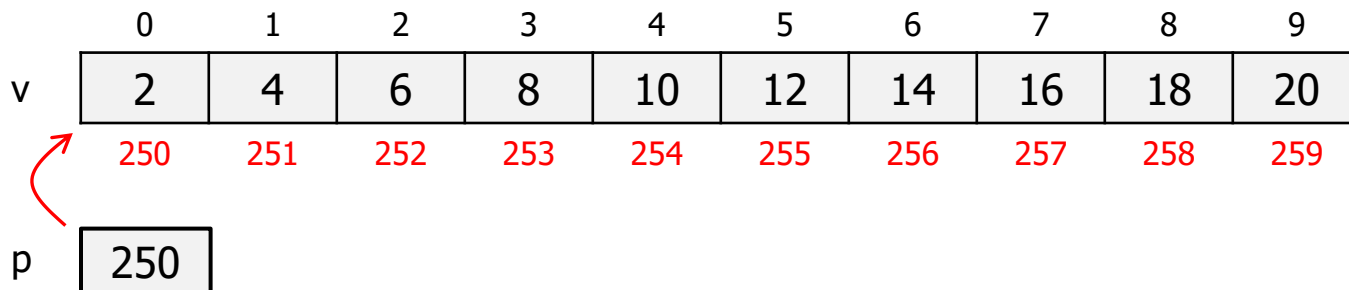
Exemplo 1:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
```

```
int *p;
```

```
p = v;
```

Quando usamos o nome do vetor **sem definir o índice**, na realidade estamos pegando o endereço do vetor.





Aritmética com Ponteiros

Exemplo 1:

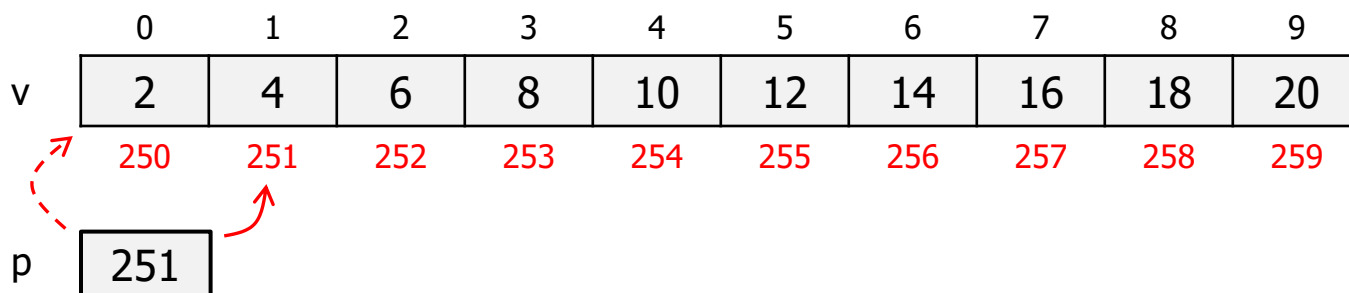
```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
```

```
int *p;
```

```
p = v;
```

```
p++;
```

Quando **incrementamos** um ponteiro, ele passa a referenciar a área de memória imediatamente **posterior**, de acordo com o tipo do ponteiro.





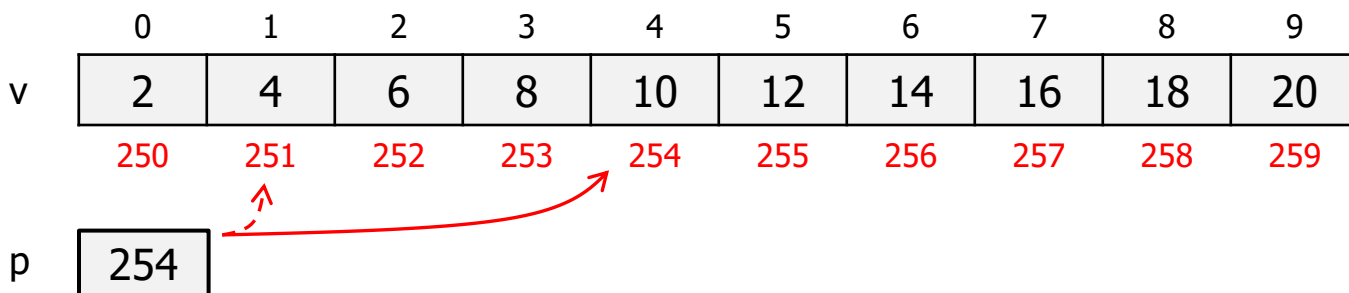
Aritmética com Ponteiros

Exemplo 1:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };  
int *p;
```

```
p = v;  
p++;  
p = p + 3;
```

Quando **somamos** um valor inteiro ao ponteiro, ele **avança** para uma posição posterior à memória atual.





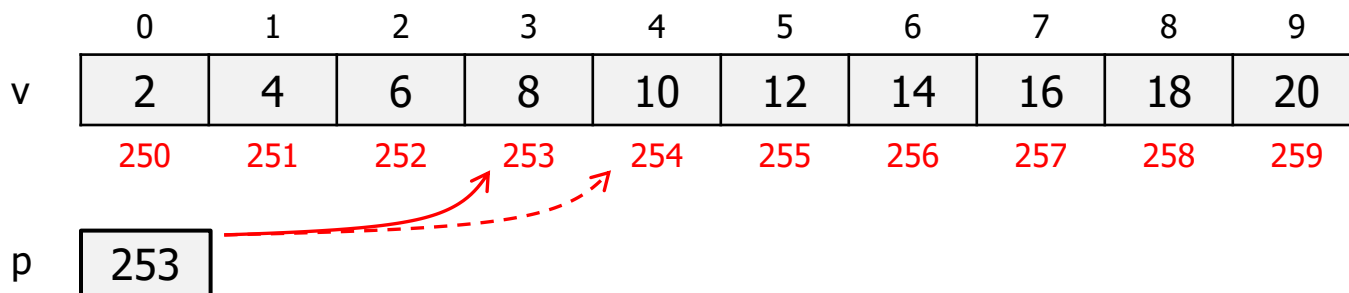
Aritmética com Ponteiros

Exemplo 1:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };  
int *p;
```

```
p = v;  
p++;  
p = p + 3;  
p--;
```

Quando **decrementamos** um ponteiro, ele passa a referenciar a área de memória imediatamente **anterior**, de acordo com o tipo do ponteiro.

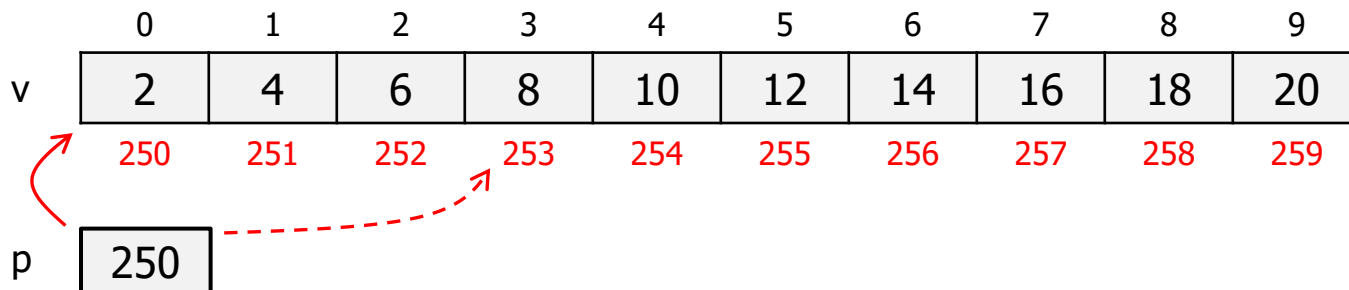




Exemplo 1:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };  
int *p;  
  
p = v;  
p++;  
p = p + 3;  
p--;  
p = p - 3;
```

Quando **subtraímos** um valor inteiro do ponteiro, ele **retrocede** para uma posição anterior à memória atual.





Aritmética com Ponteiros

Exemplo 2:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };  
int *p1, *p2, a, b;
```

	0	1	2	3	4	5	6	7	8	9
v	2	4	6	8	10	12	14	16	18	20
	250	251	252	253	254	255	256	257	258	259



Aritmética com Ponteiros

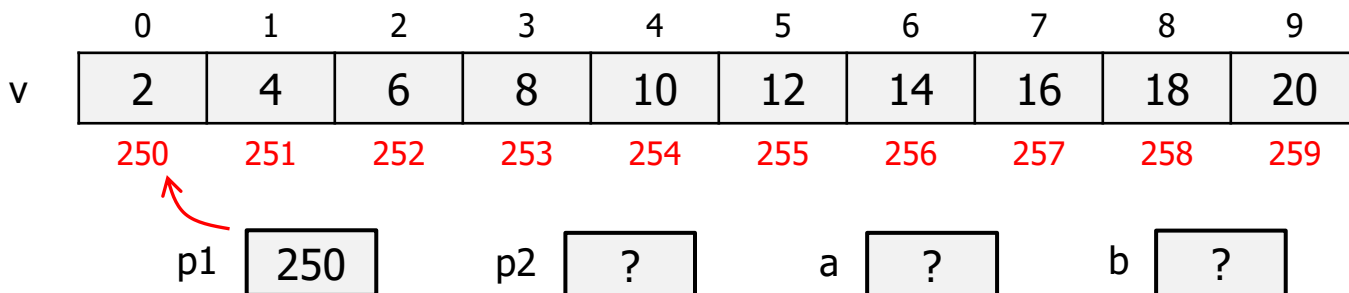
Exemplo 2:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
```

```
int *p1, *p2, a, b;
```

```
p1 = v;
```

p1 aponta para o primeiro elemento de **v**.





Aritmética com Ponteiros

Exemplo 2:

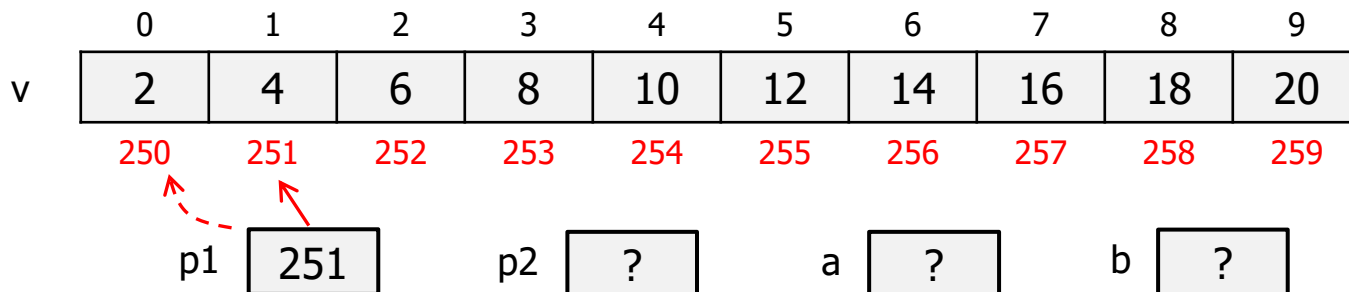
```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
```

```
int *p1, *p2, a, b;
```

```
p1 = v;
```

```
p1++;
```

p1 aponta para o próximo inteiro, ou seja, **v[1]**.





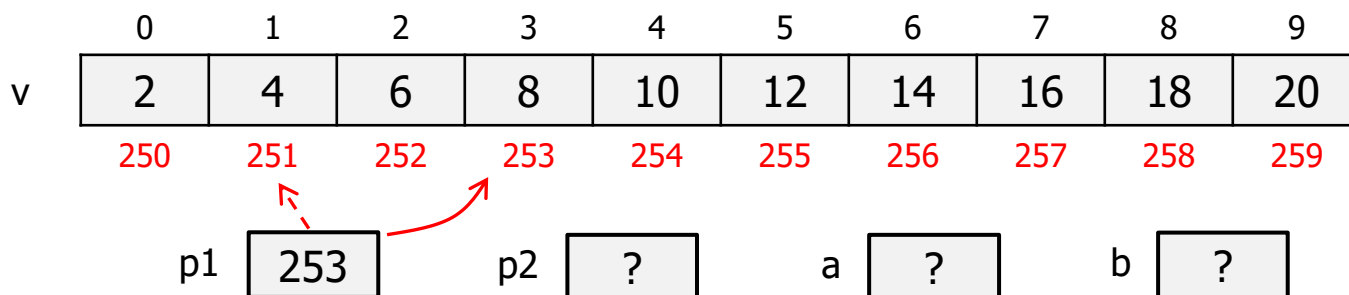
Aritmética com Ponteiros

Exemplo 2:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };  
int *p1, *p2, a, b;
```

```
p1 = v;  
p1++;  
p1 = p1 + 2;
```

p1 avança mais dois
inteiros, ou seja, **v[3]**.





Aritmética com Ponteiros

Exemplo 2:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
```

```
int *p1, *p2, a, b;
```

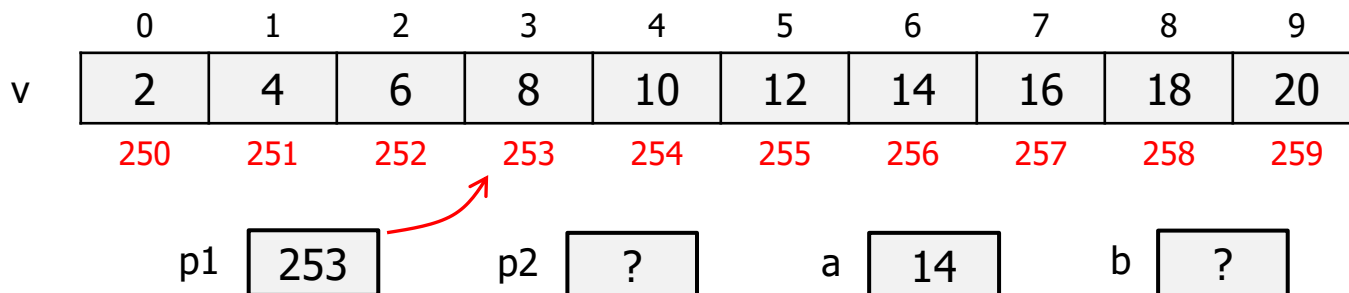
```
p1 = v;
```

```
p1++;
```

```
p1 = p1 + 2;
```

```
a = *(p1 + 3);
```

(p1 + 3) aponta para **v[6]**,
ou seja, **a = 14**.

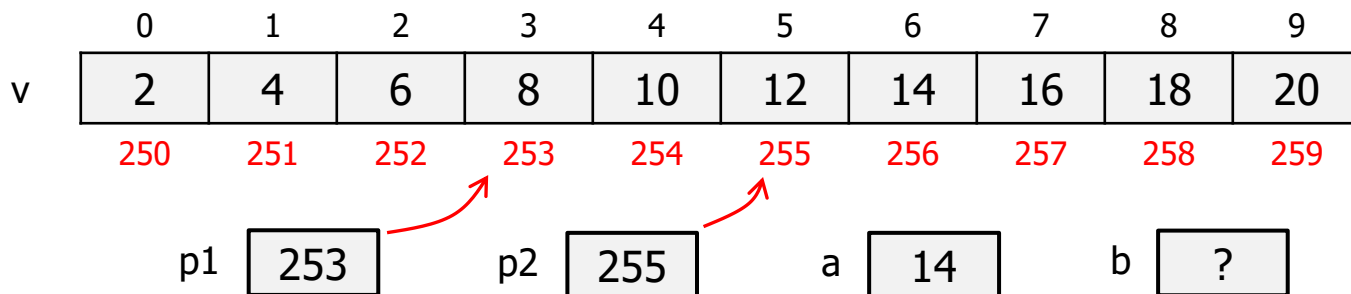




Exemplo 2:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };  
int *p1, *p2, a, b;  
  
p1 = v;  
p1++;  
p1 = p1 + 2;  
a = *(p1 + 3);  
p2 = p1 + 2;
```

p2 aponta para v[5].





Exemplo 2:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
```

```
int *p1, *p2, a, b;
```

```
p1 = v;
```

```
p1++;
```

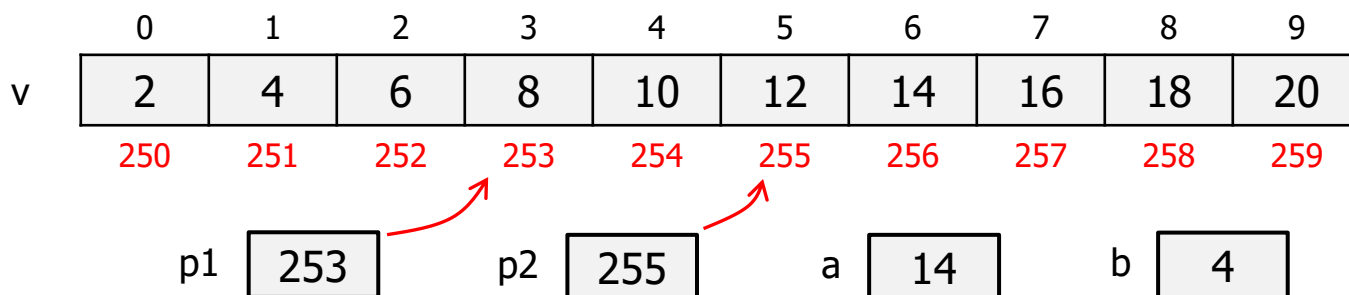
```
p1 = p1 + 2;
```

```
a = *(p1 + 3);
```

```
p2 = p1 + 2;
```

```
b = *(p2 - 4);
```

(p2 - 4) aponta para **v[1]**, ou seja, **b = 4**.





12.3) Qual o conteúdo do vetor v ao final da execução do trecho abaixo?

```
           0   1   2   3   4   5   6   7   8   9   10   11
int v[] = { 3, 6, 1, 8, 10, 4, 8, 9, 2, 12, 5, 15 };
int *p1, *p2;

p1 = v;
p2 = &v[4];
*p1 = *(p1+1)+1;
*p2 = *p2 - *p1;
*p1 += 2 * *p2
*(p2+1) = *p2 + 1;
p1++;
p2--;
*p1 = *p2 * *p2;
```



12.3) Qual o conteúdo do vetor v ao final da execução do trecho abaixo?

```
int v[] = { 3, 6, 1, 8, 10, 4, 8, 9, 2, 12, 5, 15 };  
int *p1, *p2;
```

```
p1 = v;  
p2 = &v[4];  
*p1 = *(p1+1)+1;  
*p2 = *p2 - *p1;  
*p1 += 2 * *p2  
*(p2+1) = *p2 + 1;  
p1++;  
p2--;  
*p1 = *p2 * *p2;
```

v = { 13, 64, 1, 8, 3, 4, 8, 9, 2, 12, 5, 15 }



Observe o código a seguir:

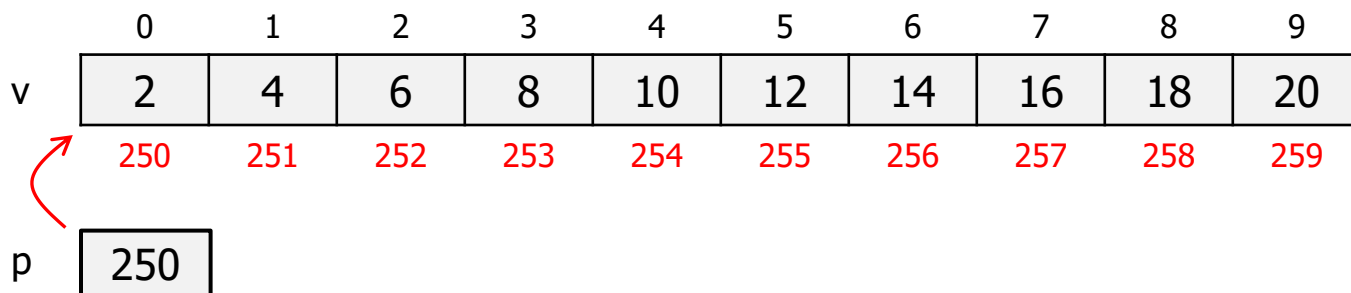
```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };  
int *p, a;
```

```
p = v;
```

```
a = *(p + 2);    // mesma coisa que: a = v[2]
```

```
*(p + 3) = 10;   // mesma coisa que: v[3] = 10;
```

***(p+2) é equivalente a v[2]**





Observe o código a seguir:

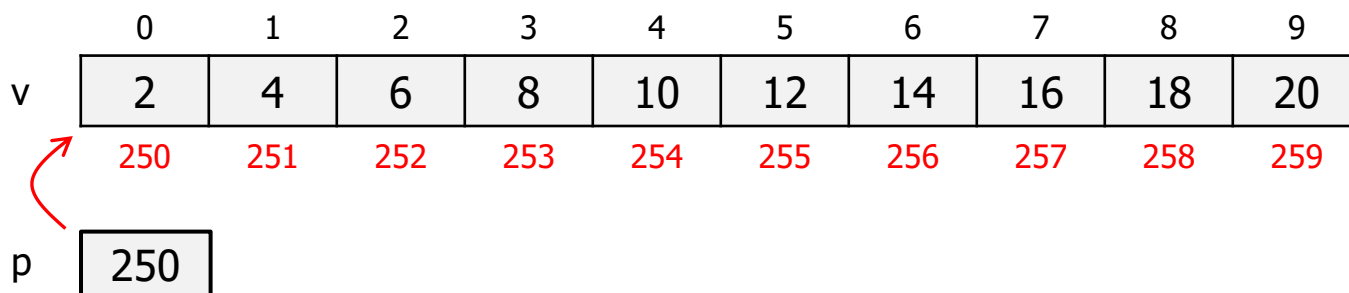
```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };  
int *p, a;
```

```
p = v;
```

```
a = *(p + 2);    // mesma coisa que: a = v[2]
```

```
*(p + 3) = 10;   // mesma coisa que: v[3] = 10;
```

***(p+3) é equivalente a v[3]**





Generalizando, dado um vetor **v** e um ponteiro **p** para esse vetor, podemos dizer que:

$*(p + i)$ é equivalente a $v[i]$

Então, as quatro formas abaixo são equivalentes:

**$v[i]$
 $*(p + i)$
 $p[i]$
 $*(v + i)$**



Esses trechos de código são totalmente equivalentes:

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
int *p;

p = v;
for (int i = 0; i < 10; i++) {
    printf("%d - %d\n", *(p + i), *(v + i));
}
```

```
int v[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
int *p;

p = v;
for (int i = 0; i < 10; i++) {
    printf("%d - %d\n", p[i], v[i]);
}
```




Também é possível definir ponteiros para **structs**:

```
nome_da_estrutura *ponteiro;
```



Ponteiro para Struct

Exemplo:

```
struct Funcionario {  
    int matricula;  
    float salario;  
    int num_deps;  
};  
  
int main() {  
    Funcionario func;  
    Funcionario *ptr;  
  
    ptr = &func;  
    ptr->matricula = 243564;  
    ptr->salario = 5450.0;  
    ptr->num_deps = 2;  
}
```

ptr



func

matricula

?

salario

?

num_deps

?




Ponteiro para Struct

Exemplo:

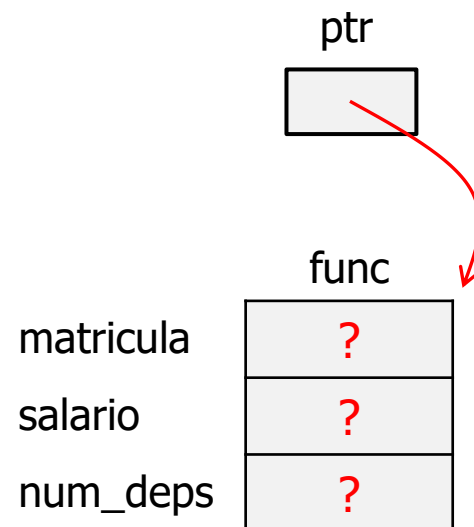
```
struct Funcionario {  
    int matricula;  
    float salario;  
    int num_deps;  
};
```

```
int main() {  
    Funcionario func;  
    Funcionario *ptr;
```

ptr aponta para func.



```
    ptr = &func;  
    ptr->matricula = 243564;  
    ptr->salario = 5450.0;  
    ptr->num_deps = 2;  
}
```





Ponteiro para Struct

Exemplo:

```
struct Funcionario {  
    int matricula;  
    float salario;  
    int num_deps;  
};  
  
int main() {  
    Funcionario func;  
    Funcionario *ptr;  
  
    ptr = &func;  
    ptr->matricula = 243564;  
    ptr->salario = 5450.0;  
    ptr->num_deps = 2;  
}
```

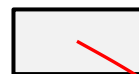
Quando acessamos os campos da struct com uma **variável** usamos:

variável.campo

Entretanto, quando acessamos os campos da struct com um **ponteiro** usamos:

ponteiro->campo

ptr



func

matricula

243564

salario

?

num_deps

?



Ponteiro para Struct

Exemplo:

```
struct Funcionario {  
    int matricula;  
    float salario;  
    int num_deps;  
};  
  
int main() {  
    Funcionario func;  
    Funcionario *ptr;  
  
    ptr = &func;  
    ptr->matricula = 243564;  
    ptr->salario = 5450.0;  
    ptr->num_deps = 2;  
}
```

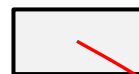
Quando acessamos os campos da struct com uma **variável** usamos:

variável.campo

Entretanto, quando acessamos os campos da struct com um **ponteiro** usamos:

ponteiro->campo

ptr



func

matricula

243564

salario

5450.0

num_deps

?



Ponteiro para Struct

Exemplo:

```
struct Funcionario {  
    int matricula;  
    float salario;  
    int num_deps;  
};  
  
int main() {  
    Funcionario func;  
    Funcionario *ptr;  
  
    ptr = &func;  
    ptr->matricula = 243564;  
    ptr->salario = 5450.0;  
    ptr->num_deps = 2;  
}
```

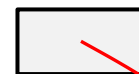
Quando acessamos os campos da struct com uma **variável** usamos:

variável.campo

Entretanto, quando acessamos os campos da struct com um **ponteiro** usamos:

ponteiro->campo

ptr



func

matricula

243564

salario

5450.0

num_deps

2



Passagem por Referência

Estudamos anteriormente a passagem de parâmetros por **valor** e por **referência**.

No C/C++, a passagem de parâmetros por referência também pode ser feita usando **ponteiros**.



Passagem por Referência

Exemplo:

```
void troca(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

Note que os parâmetros **p1** e **p2** **não são inteiros**, eles são **ponteiros para inteiros**.

```
int main(void)
{
    int a = 10, b = 20;

    troca(&a, &b);

    printf("%d  %d\n", a, b);
}
```




Passagem por Referência

Exemplo:

```
void troca(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main(void)
{
    int a = 10, b = 20;

    troca(&a, &b);

    printf("%d  %d\n", a, b);
}
```

a

10

b

20



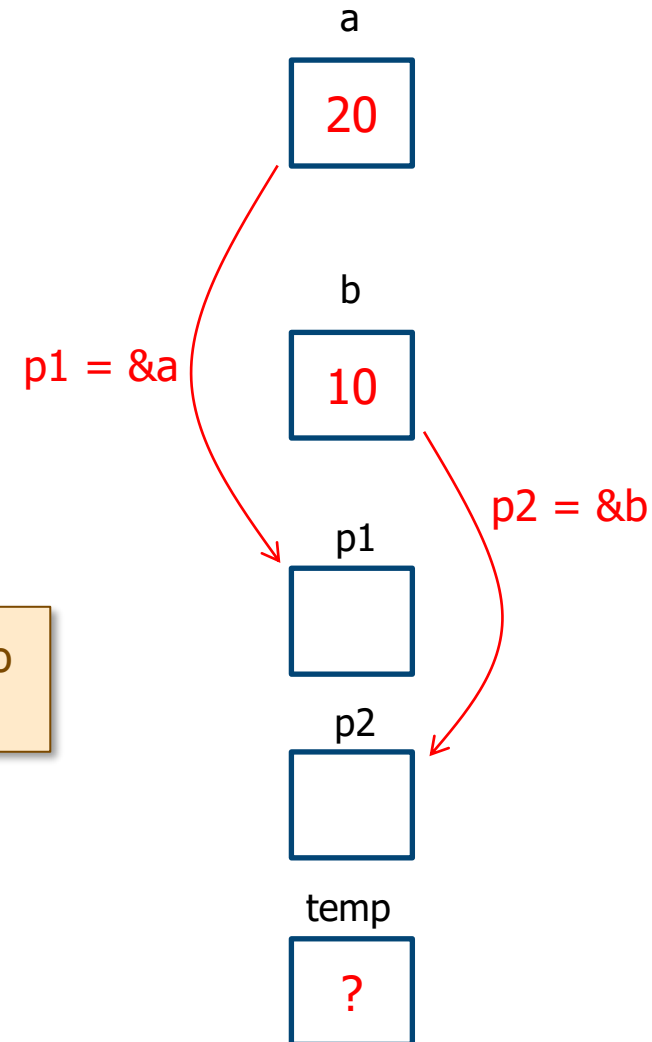
Passagem por Referência

Exemplo:

```
void troca(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main(void)
{
    int a = 10, b = 20;
    troca(&a, &b);
    printf("%d  %d\n", a, b);
}
```

Passa para a função o endereço de a e b.





Passagem por Referência

Exemplo:

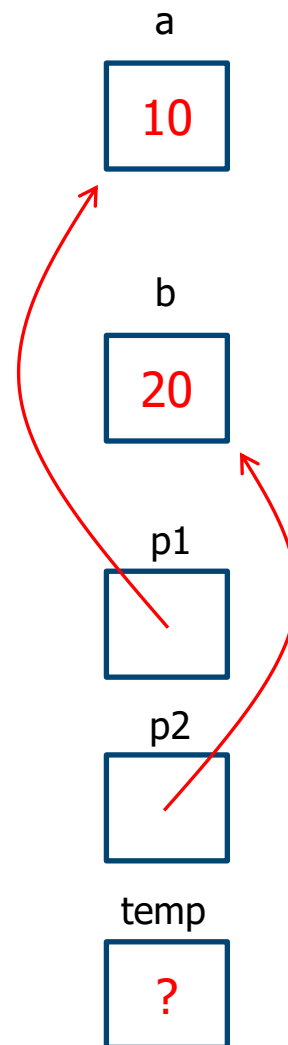
```
void troca(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main(void)
{
    int a = 10, b = 20;

    troca(&a, &b);

    printf("%d  %d\n", a, b);
}
```

Assim:
p1 aponta para a
p2 aponta para b





Passagem por Referência

Exemplo:

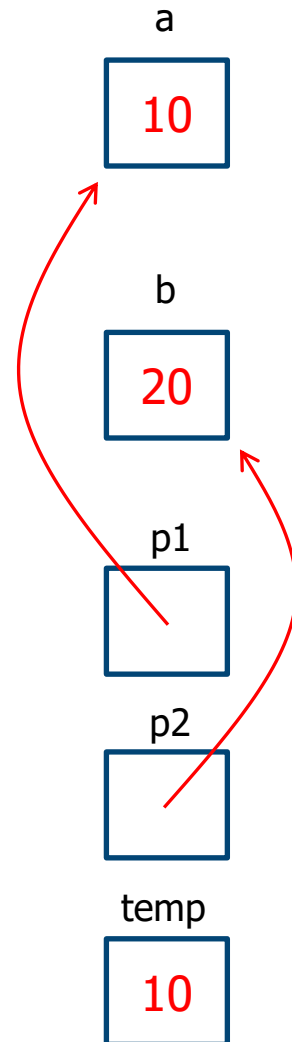
```
void troca(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main(void)
{
    int a = 10, b = 20;

    troca(&a, &b);

    printf("%d  %d\n", a, b);
}
```

temp recebe o valor apontado por **p1**.





Passagem por Referência

Exemplo:

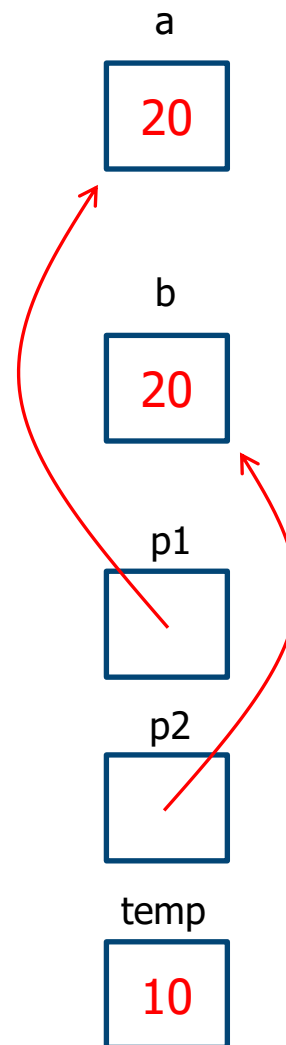
```
void troca(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main(void)
{
    int a = 10, b = 20;

    troca(&a, &b);

    printf("%d  %d\n", a, b);
}
```

Valor apontado por **p1**
recebe valor apontado
por **p2**.





Passagem por Referência

Exemplo:

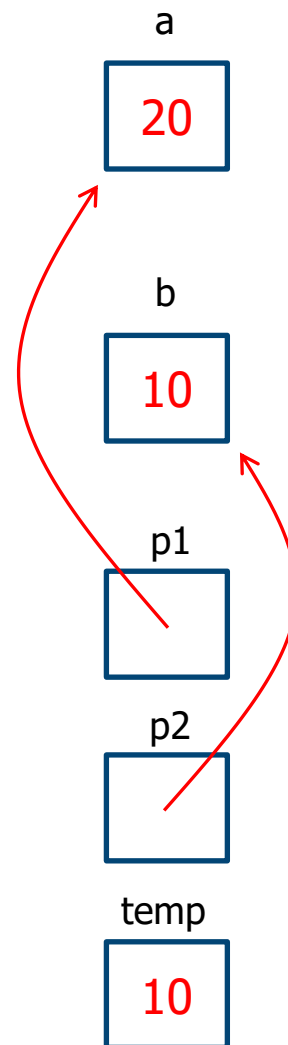
```
void troca(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main(void)
{
    int a = 10, b = 20;

    troca(&a, &b);

    printf("%d  %d\n", a, b);
}
```

Valor apontado por
p2 recebe **temp**.





Passagem por Referência

Exemplo:

```
void troca(int *p1, int *p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main(void)
{
    int a = 10, b = 20;

    troca(&a, &b);

    printf("%d  %d\n", a, b);
}
```

a

20

b

10

A função **troca** termina e o valor de **a** foi trocado com **b**.



Alocação Dinâmica de Memória

Às vezes precisamos de (grandes) áreas de memória para executar algumas tarefas dentro de um programa.

Nesses casos, é interessante alocar esse espaço no momento em que ele for necessário, e desalocá-lo assim que ele não for mais necessário.

Isso é possível através da **alocação dinâmica de memória**.

Mas como eu vou trabalhar com uma área de memória se eu não tenho uma variável para ela?

Usando **ponteiros!**



Foto criada por freepik - br.freepik.com



Alocação Dinâmica de Memória

Para **alocar** a memória em programas C usamos a função:

void *malloc(*long tam*)

Retorna o **endereço** da memória alocada.

Quantidade de **bytes** a serem alocados na memória.



Se não houver memória suficiente o malloc retorna NULL



Alocação Dinâmica de Memória

Para **desalocar** a memória em programas C usamos a função:

```
void free(void *ptr)
```

Ponteiro que referencia
a memória a ser
desalocada.



Alocação Dinâmica de Memória

Para definir a quantidade de bytes necessárias usamos o operador `sizeof`:

`sizeof(tipo)`

Exemplo	Descrição
<code>sizeof(char)</code>	Número de bytes ocupados por um char
<code>sizeof(int)</code>	Número de bytes ocupados por um int
<code>sizeof(long)</code>	Número de bytes ocupados por um long
<code>sizeof(float)</code>	Número de bytes ocupados por um float
<code>sizeof(double)</code>	Número de bytes ocupados por um double



Alocação Dinâmica de Memória

Também é possível aplicar o operador `sizeof` a uma `struct`:

`sizeof(nome da struct)`

```
struct Funcionario {  
    int matricula;  
    char nome[60];  
    float salario;  
};
```

`sizeof(Funcionario)` → número de bytes ocupados pela struct Funcionario



Alocação Dinâmica de Memória

Exemplo:

Para usar as funções de alocação dinâmica precisamos incluir o **stdlib.h**.

Devemos sempre usar o **cast** com a função malloc.

Para ler ou armazenar dados na área alocada usamos o operador ***** no ponteiro.

```
#include <stdlib.h>

int main() {
    int *p;
    float *q;
    char *r;

    p = (int *) malloc(sizeof(int));
    q = (float *) malloc(sizeof(float));
    r = (char *) malloc(sizeof(char));

    *p = 5;
    *q = 3.14159f;
    *r = 'A';

    printf("%d %f %c\n", *p, *q, *r);

    free(p);
    free(q);
    free(r);
}
```



Alocação Dinâmica de Memória

Para alocar um **vetor** podemos usar a função **malloc** ou **calloc**:

- ✓ **malloc**: devemos multiplicar o tamanho do vetor pelo operador `sizeof`.
- ✓ **calloc**: devemos passar o tamanho do vetor como 1º parâmetro da função.

malloc	calloc	Descrição
<code>malloc(70 * sizeof(char))</code>	<code>calloc(70, sizeof(char))</code>	Vetor de 70 caracteres
<code>malloc(10 * sizeof(int))</code>	<code>calloc(10, sizeof(int))</code>	Vetor de 10 números inteiros
<code>malloc(200 * sizeof(float))</code>	<code>calloc(200, sizeof(float))</code>	Vetor de 200 números reais
<code>malloc(15 * sizeof(Funcionario))</code>	<code>calloc(15, sizeof(Funcionario))</code>	Vetor de 15 funcionários



Alocação Dinâmica de Memória

Programas C++ também podem usar as funções `malloc` e `free`. Entretanto, existem dois novos comandos para alocar e desalocar a memória no C++: `new` e `delete`.

Os comandos `new` e `delete` tem sintaxes diferentes das funções `malloc` e `free`, mas tem exatamente o mesmo propósito.

As principais diferenças são:

- ✓ O comando `new` não precisa do `sizeof`.
- ✓ O comando `new` não precisa do `cast`.
- ✓ O comando `new` tem uma sintaxe específica para alocar vetores.



Alocação Dinâmica de Memória

O comando **new** tem duas sintaxes:

new tipo

new tipo[tam]

Aloca um vetor de tamanho **tam**.



Se não houver memória suficiente o new retorna NULL



Alocação Dinâmica de Memória

O comando **delete** tem duas sintaxes:

delete *ptr*

delete [] *ptr*

Usada para desalocar
um vetor.



Alocação Dinâmica de Memória

C

```
int *p;
char *q;
Funcionario *f;

p = (int *) malloc(sizeof(int));
q = (char *) malloc(sizeof(char));
f = (Funcionario *) malloc(sizeof(Funcionario));

free(p);
free(q);
free(f);

p = (int *) malloc(10 * sizeof(int));
q = (char *) malloc(50 * sizeof(char));
f = (Funcionario *) malloc(15 * sizeof(Funcionario));

free(p);
free(q);
free(f);
```

C++

```
int *p;
char *q;
Funcionario *f;

p = new int;
q = new char;
f = new Funcionario;

delete p;
delete q;
delete f;

p = new int[10];
q = new char[50];
f = new Funcionario[15];

delete [] p;
delete [] q;
delete [] f;
```



Importante!

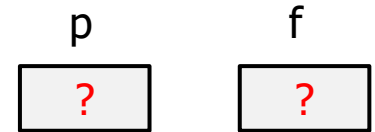
- ✓ Ao adotar a alocação dinâmica de memória devemos ter especial atenção com situações de **Memory Leak** (vazamento de memória).
- ✓ Memory leak é uma situação onde o programa aloca uma área e "perde" o ponteiro para essa área.
- ✓ Sem uma referência para a área alocada é impossível liberá-la, ou seja, o programa fica com uma área de memória que ele não pode nem usar nem liberar.
- ✓ Assim, devemos ter sempre o cuidado de liberar áreas de memória que não são mais necessárias.





Exemplo:

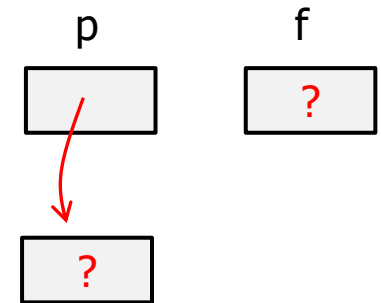
```
→ int *p;  
   Funcionario *f;  
  
   p = (int *) malloc(sizeof(int));  
   f = (Funcionario *) malloc(sizeof(Funcionario));  
  
   *p = 5;  
   f->matricula = 1357;  
   strcpy(f->nome, "Joao");  
   f->salario = 2350;  
  
   p = (int *) malloc(sizeof(int));  
   f = (Funcionario *) malloc(sizeof(Funcionario));
```





Exemplo:

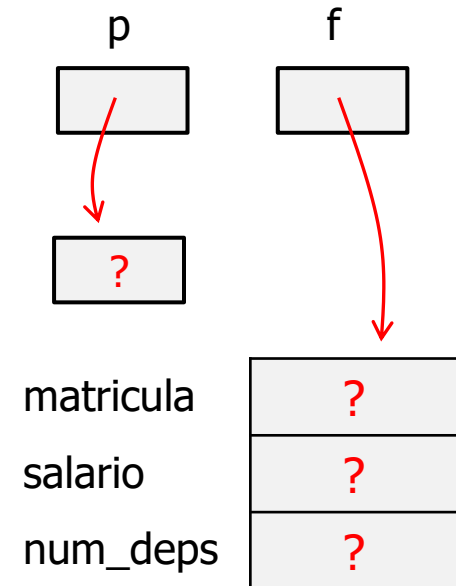
```
int *p;  
Funcionario *f;  
  
→ p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));  
  
*p = 5;  
f->matricula = 1357;  
strcpy(f->nome, "Joao");  
f->salario = 2350;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));
```





Exemplo:

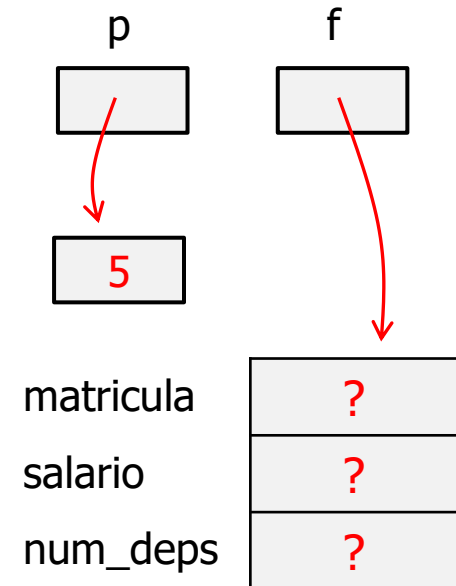
```
int *p;  
Funcionario *f;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));  
  
*p = 5;  
f->matricula = 1357;  
strcpy(f->nome, "Joao");  
f->salario = 2350;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));
```





Exemplo:

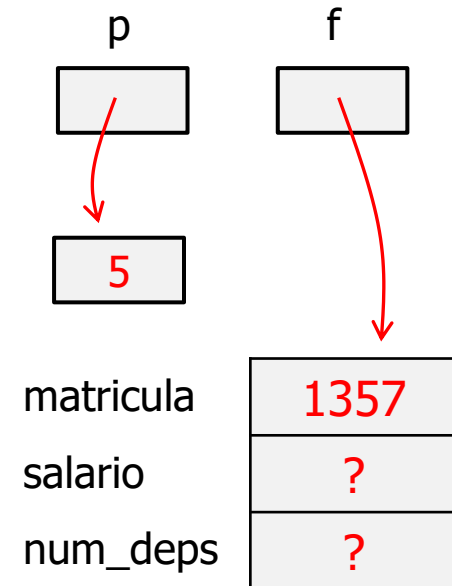
```
int *p;  
Funcionario *f;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));  
  
→ *p = 5;  
f->matricula = 1357;  
strcpy(f->nome, "Joao");  
f->salario = 2350;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));
```





Exemplo:

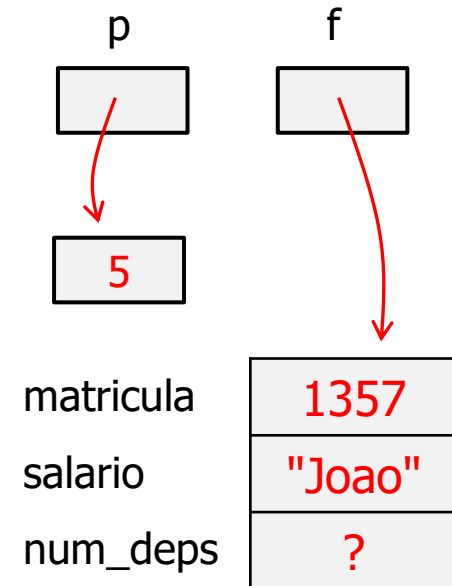
```
int *p;  
Funcionario *f;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));  
  
*p = 5;  
→ f->matricula = 1357;  
strcpy(f->nome, "Joao");  
f->salario = 2350;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));
```





Exemplo:

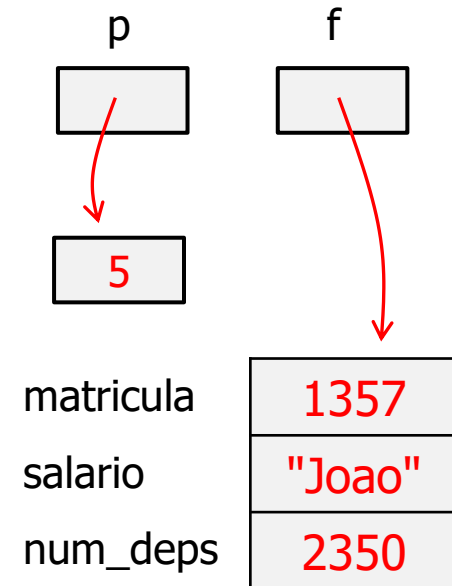
```
int *p;  
Funcionario *f;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));  
  
*p = 5;  
f->matricula = 1357;  
f->salario = 2350;  
strcpy(f->nome, "Joao");  
f->num_deps = 1;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));
```





Exemplo:

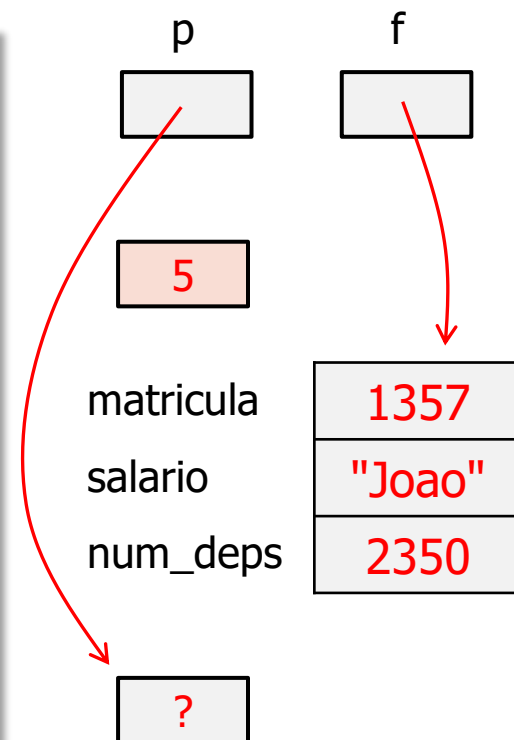
```
int *p;  
Funcionario *f;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));  
  
*p = 5;  
f->matricula = 1357;  
strcpy(f->nome, "Joao");  
f->salario = 2350;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));
```





Exemplo:

```
int *p;  
Funcionario *f;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));  
  
*p = 5;  
f->matricula = 1357;  
strcpy(f->nome, "Joao");  
f->salario = 2350;  
  
→ p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));
```





Exemplo:

```
int *p;  
Funcionario *f;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));  
  
*p = 5;  
f->matricula = 1357;  
strcpy(f->nome, "Joao");  
f->salario = 2350;  
  
p = (int *) malloc(sizeof(int));  
f = (Funcionario *) malloc(sizeof(Funcionario));
```

