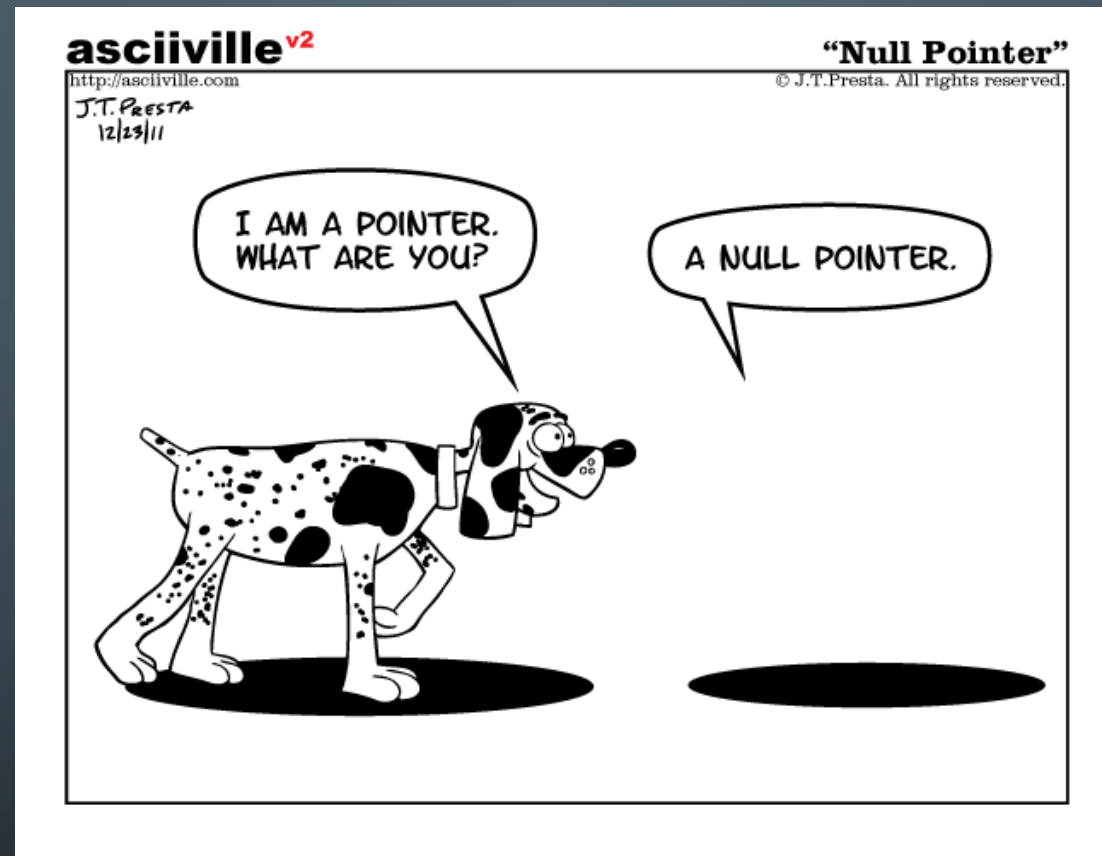


PONTEIROS EM C



- Toda informação está na **MEMÓRIA**.
- Quando criamos uma variável, é reservado um espaço em memória para armazenar o valor que queremos.

Data Type	Size (bytes)	Size (bits)	Value Range
unsigned char	1	8	0 to 255
signed char	1	8	-128 to 127
char	1	8	either
unsigned short	2	16	0 to 65,535
short	2	16	-32,768 to 32,767
unsigned int	4	32	0 to 4,294,967,295
int	4	32	-2,147,483,648 to 2,147,483,647
unsigned long	8	64	0 to 18,446,744,073,709,551,616
long	8	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	64	0 to 18,446,744,073,709,551,616
long long	8	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	32	3.4E +/- 38 (7 digits)
double	8	64	1.7E +/- 308 (15 digits)
long double	8	64	1.7E +/- 308 (15 digits)
bool	1	8	false or true

- Analisando o tipo **int**...

Data Type	Size (bytes)	Size (bits)	Value Range
int	4	32	-2,147,483,648 to 2,147,483,647

- Como é definida a coluna "Value Range"?

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(){
6
7     int a = 12;
8
9 }

```

$$(12)_{10} = (1100)_2$$

Byte 4	Byte 3	Byte 2	Byte 1
00000000	00000000	00000000	00001100

Endereço de memória	Valor
...	
207	
206	00000000
205	00000000
204	00000000
203	00001100
202	
...	

a

Um computador
que tem 4 GB de
RAM, possui
34.359.738.368
bits!



- Enfim... variáveis fazem acesso direto a uma posição de memória!
- E ponteiro? O que é isso?

“Ponteiros são um tipo especial de variáveis que permitem armazenar **endereços de memória** em vez de dados numéricos (como os tipos **int**, **float** e **double**) ou caracteres (como o tipo **char**).”

- Resumindo:

→ **Variável**: é um espaço reservado de memória usado para **guardar um valor**.

→ **Ponteiro**: é um espaço reservado de memória usado para **guardar um endereço de memória**.

Na linguagem C, um ponteiro pode ser declarado para qualquer tipo de variável (**char, int, float, double** etc.), inclusive para aquelas criadas pelo programador (**struct** etc.)!

- Como declarar um ponteiro:

tipo *nome_do_ponteiro

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    //Declara um ponteiro para int
    int *p;
    //Declara um ponteiro para float
    float *x;
    //Declara um ponteiro para char
    char *y;
    //Declara uma variável do tipo int e um ponteiro para int
    int soma, *p2,;
    system("pause");
    return 0;
}
```

○ ***** é um operador que indica ao compilador que aquela variável irá armazenar um endereço de memória, e não um valor!

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int x = 3, y = 5, z;
    z = y * x;
    int *p;

    system("pause");
    return 0;
}
```

○ * é um operador que indica ao compilador que aquela variável irá armazenar um endereço de memória, e não um valor!

- Ponteiros apontam para uma posição de memória
- Ponteiros não inicializados apontam para um lugar indefinido! QUALQUER USO DESSE PONTEIRO causa um **comportamento indefinido**
- Ponteiros devem ser inicializados
- Se não conhece o endereço de inicialização a priori, use **NULL** (0 na maioria dos computadores)

- Mas como inicializar/instanciar um ponteiro com um valor válido?
- Use o operador **&** na frente do nome da variável.
- O operador & é unário e é conhecido como operador de endereço.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    //Declara uma variável int contendo o valor 10
    int count = 10;
    //Declara um ponteiro para int
    int *p;
    //Atribui ao ponteiro o endereço da variável int
    p = &count;

    system("pause");
    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    //Declara uma variável int contendo o valor 10
    int count = 10;
    //Declara um ponteiro para int
    int *p;
    //Atribui ao ponteiro o endereço da variável int
    p = &count;

    system("pause");
    return 0;
}
```

Memória		
#	var	conteúdo
119		
120	int *p	#122
121		
122	int count	10
123		

- E como saber o valor guardado dentro de uma posição que tem um endereço de memória (ponteiro)?
- Use o operador `*` na frente do ponteiro.
- O operador `*` é unário e é conhecido como operador de indireção ou desreferenciação.


```

#include <stdio.h>
#include <stdlib.h>
int main(){
    //Declara uma variável int contendo o valor 10
    int count = 10;
    //Declara um ponteiro para int
    int *p;
    //Atribui ao ponteiro o endereço da variável int
    p = &count;
    printf("Conteudo apontado por p: %d \n",*p);
    //Atribui um novo valor à posição de memória
    apontada por p
    *p = 12;
    printf("Conteudo apontado por p: %d \n",*p);
    printf("Conteudo de count: %d \n",count);

    system("pause");
    return 0;
}

```

```

Conteudo apontado por p: 10
Conteudo apontado por p: 12
Conteudo de count: 12

```

Memória		
#	var	conteúdo
119		
120	int *p	#122
121		
122	int count	10
123		


```
#include <stdio.h>
#include <stdlib.h>
int main(){
    //Declara uma variável int contendo o valor 10
    int count = 10;
    //Declara um ponteiro para int
    int *p;
    //Atribui ao ponteiro o endereço da variável int
    p = &count;
    printf("Conteúdo apontado por p: %d \n",*p);
    //Atribui um novo valor à posição de memória
    apontada por p
    *p = 12;
    printf("Conteúdo apontado por p: %d \n",*p);
    printf("Conteúdo de count: %d \n",count);

    system("pause");
    return 0;
}
```

```
Conteúdo apontado por p: 10
Conteúdo apontado por p: 12
Conteúdo de count: 12
```

- Assim...

Como *p* aponta para *count*, e ao dizer **p* estou acessando o conteúdo apontado por *p*, então

****p = count***
(e vice versa)

Exemplo: operador "*" versus operador "&"

"*"	Declara um ponteiro: <code>int *x;</code>
	Conteúdo para onde o ponteiro aponta: <code>int y = *x;</code>
"&"	Endereço onde uma variável está guardada na memória: <code>&y</code>

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(){
6
7     int a = 12;
8     int *p;
9
10    p = &a;
11
12    printf("Conteúdo de p: %p\n",p);
13    printf("Endereço de p: %p\n",&p);
14    printf("Conteúdo da posição apontada por p: %d\n",*p);
15
16 }
```

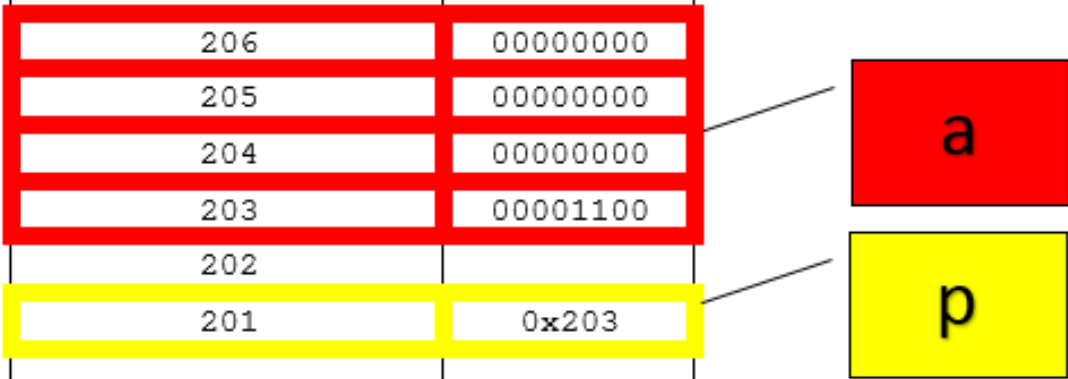
```
Running "/home/ubuntu/workspace/testes_c/ponteiros.c"
Conteúdo de p: 0x7ffee82de744
Endereço de p: 0x7ffee82de748
Conteúdo da posição apontada por p: 12

Process exited with code: 0
```

- Aritmética de ponteiros
- Operações aritméticas: adição, subtração, atribuição entre ponteiros (do mesmo tipo).
- Incrementando nosso exemplo:

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(){
6
7     int a = 12;
8     int *p;
9
10    p = &a;
11
12    printf("Conteúdo de p: %p\n",p);
13    printf("Endereço de p: %p\n",&p);
14    printf("Conteúdo da posição apontada por p: %d\n",*p);
15
16    p++;
17
```

Endereço de memória	Valor
...	
207	
206	00000000
205	00000000
204	00000000
203	00001100
202	
201	0x203
...	



```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(){
6
7     int a = 12;
8     int *p;
9
10    p = &a;
11
12    printf("Conteúdo de p: %p\n",p);
13    printf("Endereço de p: %p\n",&p);
14    printf("Conteúdo da posição apontada por p: %d\n",*p);
15
16    p++;
17
```

Qual será o novo valor de p
após este comando (lembrando
que p tinha o valor 0x203)?

- Ponteiros têm um relacionamento íntimo com arrays!
- Arrays são grupos do mesmo tipo de dado, organizados na memória de forma contígua!

```
91 int exemplo05(){
92
93     int numeros[7] = {10,30,2,4,8,22,50};
94     int *ptr_numeros = numeros;
95
96     for (int i = 0; i < 7; i++) {
97         printf("%d\t",numeros[i]);
98     }
99
100    printf("\n ----- imprimindo agora os endereços ----- \n");
101
102    for (int c = 0; c < 7; c++) {
103        printf("%p\t",ptr_numeros);
104        ptr_numeros++;
105    }
106 }
```

0	1	2	3	4	5	6
10	30	2	4	8	22	50
1000	1004	1008	1012	1016	1020	1024

```
91 int exemplo05(){
92
93     int numeros[7] = {10,30,2,4,8,22,50};
94     int *ptr_numeros = numeros;
95
96     for (int i = 0; i < 7; i++) {
97         printf("%d\t",numeros[i]);
98     }
99
100     printf("\n ----- imprimindo agora os endereços ----- \n");
101
102     for (int c = 0; c < 7; c++) {
103         printf("%p\t",ptr_numeros);
104         ptr_numeros++;
105     }
106 }
```

```
107 int exemplo06(){
108
109     int numeros[7] = {10,30,2,4,8,22,50};
110     int *ptr_numeros = numeros;
111
112     for (int i = 0; i < 7; i++) {
113         printf("%d\t",numeros[i]);
114     }
115
116     printf("\n ----- imprimindo agora os endereços ----- \n");
117
118     for (; ptr_numeros != numeros+7; ptr_numeros++) {
119         printf("%p\t",ptr_numeros);
120     }
121 }
```



```
91 int exemplo05(){
92
93     int numeros[7] = {10,30,2,4,8,22,50};
94     int *ptr_numeros = numeros;
95
96     for (int i = 0; i < 7; i++) {
97         printf("%d\t",numeros[i]);
98     }
99
100    printf("\n ----- imprimindo agora os endereços ----- \n");
101
102    for (int c = 0; c < 7; c++) {
103        printf("%p\t",ptr_numeros);
104        ptr_numeros++;
105    }
106 }
```

```
123 int exemplo07(){
124     int numeros[7] = {10,30,2,4,8,22,50};
125     int *ptr_numeros = numeros;
126
127     for (; ptr_numeros != numeros+7; ptr_numeros++) {
128         printf("%d\t",*ptr_numeros);
129     }
130
131     printf("\n ----- imprimindo agora os endereços ----- \n");
132
133     for (ptr_numeros=numeros; ptr_numeros != numeros+7; ptr_numeros++) {
134         printf("%p\t",ptr_numeros);
135     }
136 }
```

- E como ficam os ponteiros com arrays multidimensionais?

```
158 int exemplo09(){
159     int mat[2][2] = {{1,2},{3,4}};
160     int i,j;
161     printf("----- imprimindo arrays multidimensionais da forma tradicional -----\\n");
162     for(i=0;i<2;i++)
163         for(j=0;j<2;j++)
164             printf("%d\\n", mat[i][j]);
165
166     printf("----- outra forma, usando ponteiros -----\\n");
167
168     int *p = &mat[0][0];
169     for(i=0;i<4;i++)
170         printf("%d\\n", *(p+i));
171     return 0;
172 }
```

- Voltando às funções...
- Em C, todos os parâmetros são passados às funções por valor (cópia).
- Usando ponteiros podemos fazer a passagem de parâmetros por referência.
- Na chamada da função, sempre deve-se usar o operador & para parâmetros do tipo ponteiro.

```
174 void Troca(int*a,int*b){
175     int temp;
176     temp = *a;
177     *a = *b;
178     *b = temp;
179 }
180
181 int exemplo10(int *n){
182     *n=*n+1; //ou poderia ser (*n)++
183 }
184
185 int exemplo11(){
186     int x = 1;
187     int y = 3;
188     //chamando exemplo10 para incrementar 1 em x
189     exemplo10(&x);
190
191     printf("Antes: %d e %d\n",x,y);
192     Troca(&x,&y);
193     printf("Depois: %d e %d\n",x,y);
194     return 0;
195 }
```

- Passagem de vetor como parâmetro...
- É necessário declarar o próprio vetor e um segundo parâmetro que indica o seu tamanho
- São sempre passados por referência

```
197 void imprime (int *n, int m){
198     int i;
199     for (i=0; i<m;i++)
200         printf("%d \t", n[i]);
201 }
202
203 int exemplo12(int *n){
204     int v[5] = {1,2,3,4,5};
205     imprime(v,5);
206     return 0;
207 }
```

→ Note que passamos o vetor sem colchetes. Significa que estamos passando o vetor inteiro.

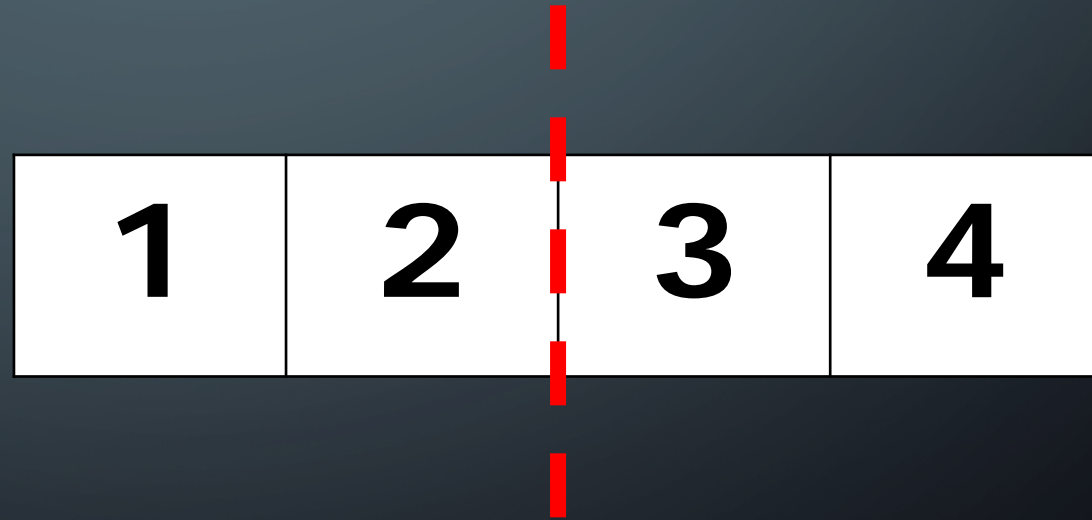
→ Note também que não precisamos do operador &, pois o próprio nome do vetor já é um apontador para a primeira posição

- Passagem de matrizes como parâmetro...
- É necessário declarar a própria matriz e o tamanho de todas as dimensões, exceto a primeira.

```
213 void imprime_matriz(int m[][2], int n){
214     int i,j;
215     for (i=0; i<n;i++){
216         for (j=0; j<2;j++){
217             printf("%d \t", m[i][j]);
218             printf("\n");
219         }
220     }
221
222     int exemplo13(){
223         int mat[3][2] = {{1,2},{3,4},{5,6}};
224         imprime_matriz(mat,3);
225         return 0;
226     }
```


- A razão disso vem da forma em que matrizes de C são representadas na memória. As linhas são colocadas em posições contíguas de memória!
- Por exemplo, a matriz 2x2 abaixo é representada assim:

1	2
3	4



- Quando você acessa o campo (i, j) da matriz 2×2 , o que o C faz é acessar o campo $2*i + j$ do "vetor".
- Por isso, o compilador tem que saber em tempo de compilação quanta colunas há em cada linha, que é o fator que multiplica o i !
- A linguagem C não faz nenhum teste de verificação dos índices usados para acessar os elementos de um vetor/matriz. Isto significa que, se estes limites não forem respeitados, resultados indesejados serão obtidos. 😊

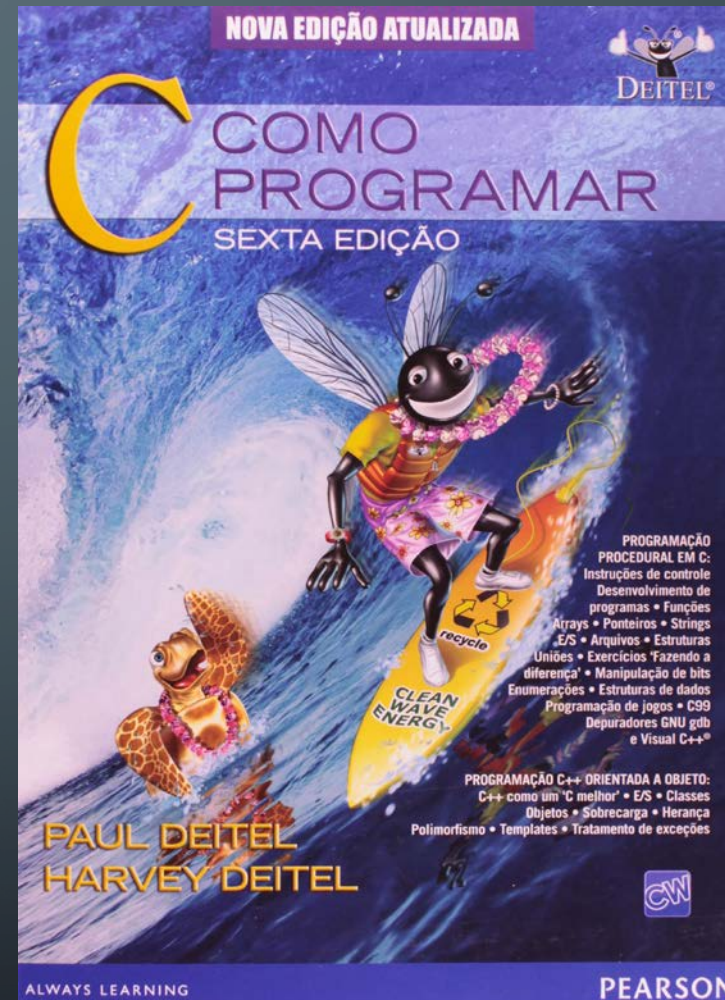
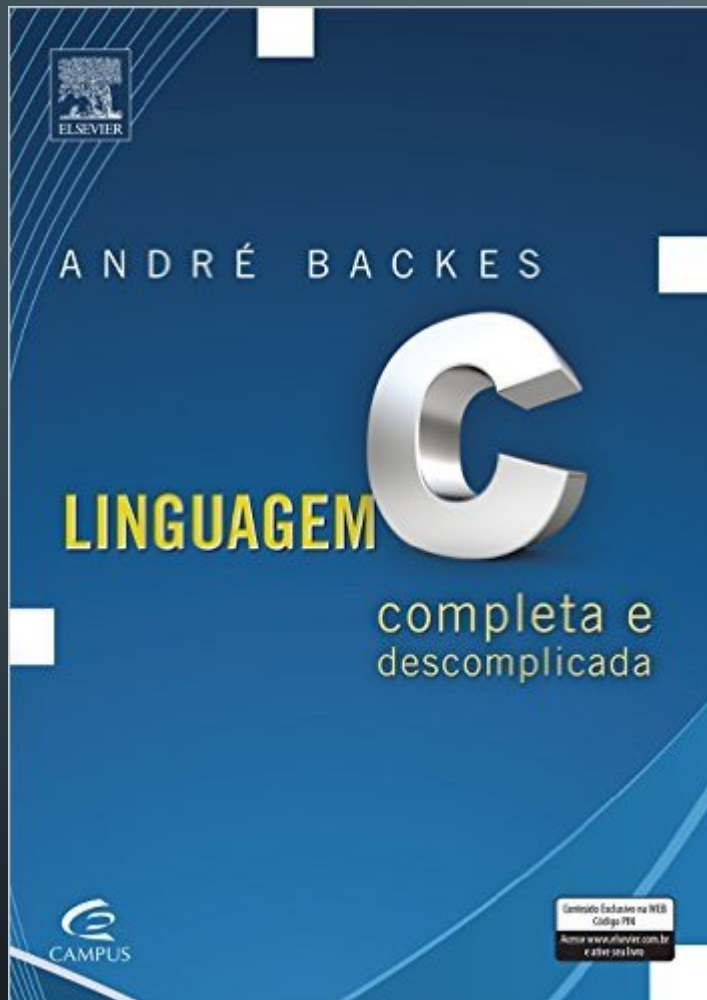
- Passagem de structs como parâmetro por valor e referência...

```
6 struct ponto {  
7     int x, y;  
8 };
```

```
237 void imprime_struct_valor(struct ponto p){  
238     printf("x = %d\n",p.x);  
239     printf("y = %d\n",p.y);  
240 }  
241  
242 int exemplo14(){  
243     struct ponto p1 = {10,20};  
244     imprime_struct_valor(p1);  
245     return 0;  
246 }
```

```
248 void instancia_struct_referencia(struct ponto *p){  
249     (*p).x = 10;  
250     (*p).y = 20;  
251 }  
252  
253 int exemplo15(){  
254     struct ponto p1;  
255     instancia_struct_referencia(&p1);  
256     printf("x = %d\n",p1.x);  
257     printf("y = %d\n",p1.y);  
258     return 0;  
259 }
```


- Referências:



- Excelente lista de videoaulas de ponteiros:

<https://www.youtube.com/playlist?list=PLa75BYTPDNKbhUVggmU3JUEBPibvh0C2t>