Algoritmos e estruturas de Dados A05

Ponteiros

Prof. Mauri Ferrandin

Ponteiros

- São variáveis cujos valores são endereços de memória de outras variáveis;
- Variáveis referenciam valores diretamente;
- Ponteiros referenciam valores indiretamente (“apontam”)
- Em C são utilizados tipicamente em:
  - Vetores/Strings
  - Passagem de argumentos a funções por referência
  - Alocação dinâmica de memória
  - Para retornar mais de um valor em uma função.
  - Referência para listas, pilhas, árvores e grafos.

Ponteiros

Sintaxe de declaração de um ponteiro

`tipo *nome_ponteiro`

- **tipo**: é o tipo da variável para a qual ele aponta
- ***** : (dereferenciador) indica que a variável será um ponteiro
- **nome_ponteiro**: O nome da variável ponteiro

```
...
int *nSensores_ptr;
float *dist_ptr;
...
float a,b,c,*d,e;
...
```

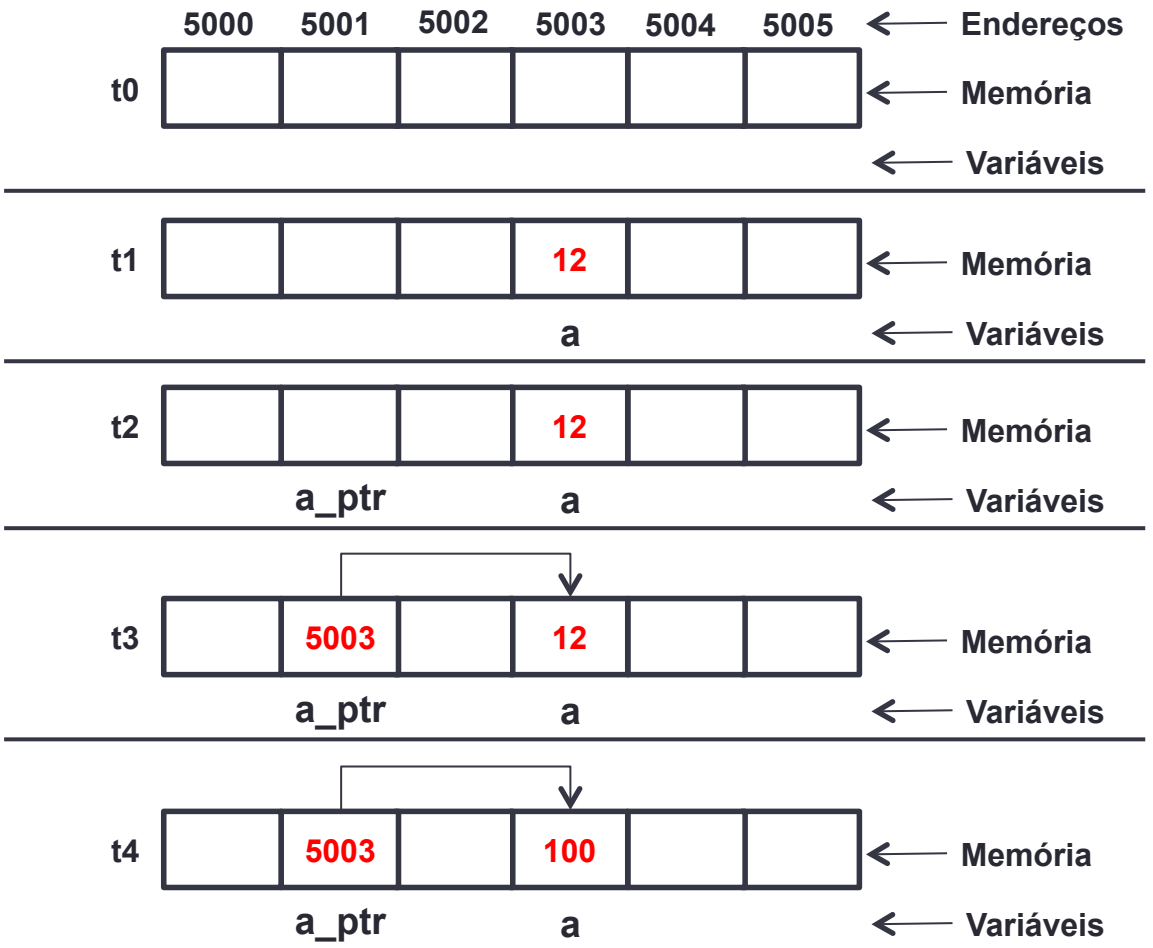
Operador de referência:

- Permite retorna o endereço na memória de seu operando
- **&** : retorna o endereço da variável;

Sintaxe:

&nome_variável

```
...
int  a = 12;   // t1
int *a_ptr;    // t2
a_ptr = &a;    // t3
*a_ptr = 100; // t4
...
```



Ponteiros - Dica

Operadores:

- Ao interpretar um programa fazer as associações:
  - **&**: o endereço de memória da variável;
  - ***: o conteúdo do endereço de memória indicado pela variável (ponteiro);

```c
#include <stdio.h>
int main(){
  int a = 10;
  int *a_ptr = &a;
  printf("valor estatico de a: %d\n",a);
  printf("endereco de memoria onde esta o valor de a:%\n",&a);
  printf("endereco de memoria armazenado no ponteiro a_ptr:
       %p\n",a_ptr);
  printf("conteudo do endereco de memoria armazenado em a_ptr:
       %d\n",*a_ptr);
  printf("conteudo do endereco de memoria armazenado da
       variavel a: %d\n",*(&a));
}
```

```
valor estatico de a: 10
endereco de memoria onde esta o valor de a: 000000000022FE44
endereco de memoria armazenado no ponteiro a_ptr: 000000000022FE44
conteudo do endereco de memoria armazenado em a_ptr: 10
conteudo do endereco de memoria armazenado da variavel a: 10
```

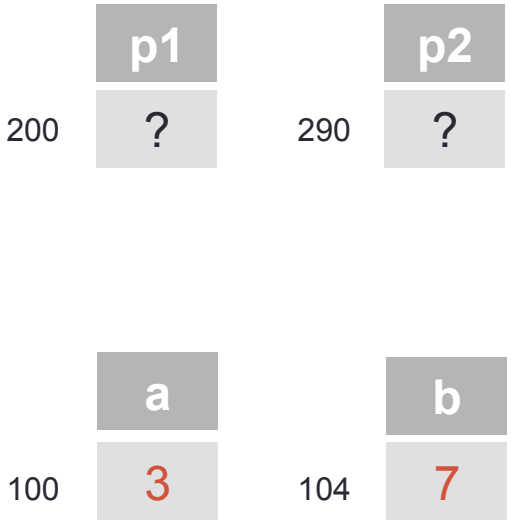
Ponteiros – exemplo 1

```c
#include <stdio.h>

int main(){
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d",*p1,*p2);
}
```

O que imprime e qual o estado das variáveis em cada ponto de execução do código?

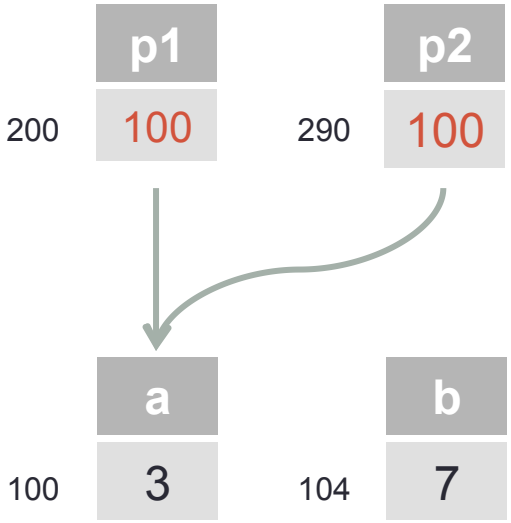
Ponteiros – exemplo 1

```c
#include <stdio.h>

int main(){
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d",*p1,*p2);
}
```



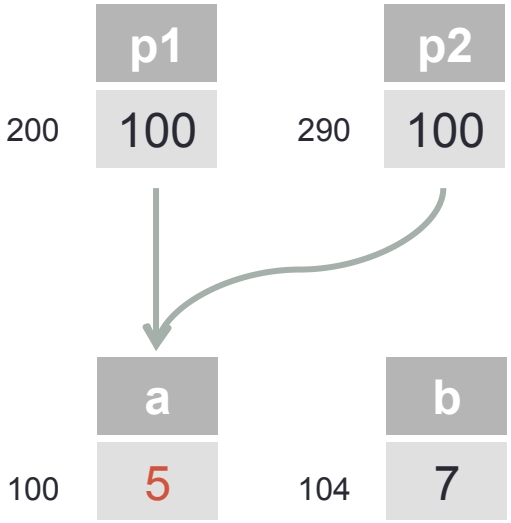
Ponteiros – exemplo 1

```c
#include <stdio.h>

int main(){
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d",*p1,*p2);
}
```



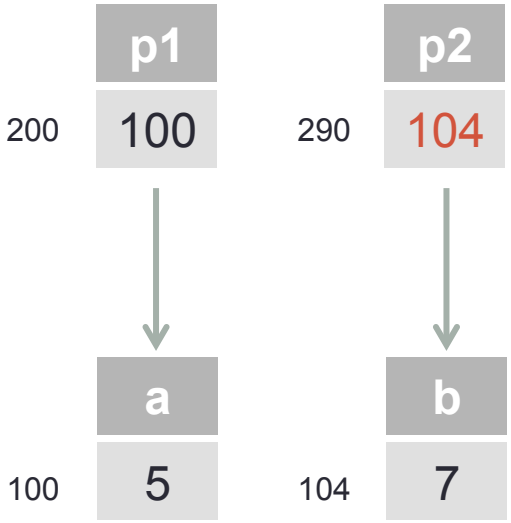
Ponteiros – exemplo 1

```c
#include <stdio.h>

int main(){
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d",*p1,*p2);
}
```



Ponteiros – exemplo 1

```c
#include <stdio.h>

int main(){
    int a = 3, b = 7;
    int *p1, *p2;
    p2 = p1 = &a;
    *p1 = 5;
    p2 = &b;
    printf("%d-%d",*p1,*p2);
}
```



Ponteiros como parâmetros para funções

Passagem por valor:

- Ao longo da função, o parâmetro recebido é uma cópia da variável recebida;
- Consequência: alterações feitas na variável passada como parâmetro, no decorrer da função, **não afetam** a variável no programa principal;

Passagem por referência (com ponteiros):

- O parâmetro na função é um ponteiro para o endereço de memória onde está o valor do parâmetro;
- Consequência: alterações feitas na variável (ponteiro) passada como parâmetro, no decorrer da função, **afetam** a variável no programa principal;

```c
#include <stdio.h>

void swap1(int x, int y){
   int z = x;
   x = y;
   y = z;
}

void swap2(int *x, int *y){
   int z = *x;
   *x = *y;
   *y = z;
}

int main(){
   int a = 10, b = 20;
   swap1(a,b);
   printf("Valor de a-b: %d - %d\n",a,b);
   swap2(&a,&b);
   printf("Valor de a-b: %d - %d\n",a,b);
}
```

x e y são variáveis locais da função e possuem uma cópia do valor de a e b **(parâmetros passados por valor)**

x e y são ponteiros com o endereço de memória das variáveis a e b **(parâmetros passados por referência)**

Saída:
Valor de a-b: 10 - 20
Valor de a-b: 20 - 10

Ponteiros – exemplo 2

```c
#include <stdio.h>

void func(int a, int *x){
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main(){
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d",a,b);
}
```

O que imprime e qual o estado das variáveis em cada ponto de execução do código?

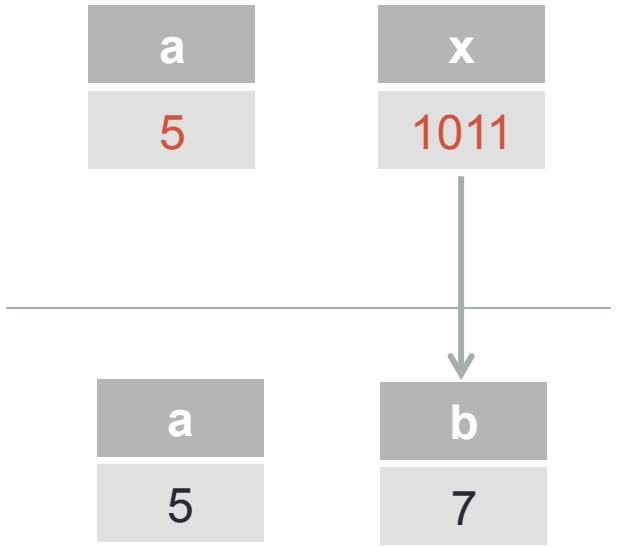
Ponteiros – exemplo 2

```c
#include <stdio.h>

void func(int a, int *x){
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main(){
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d",a,b);
}
```

a	b
5	7

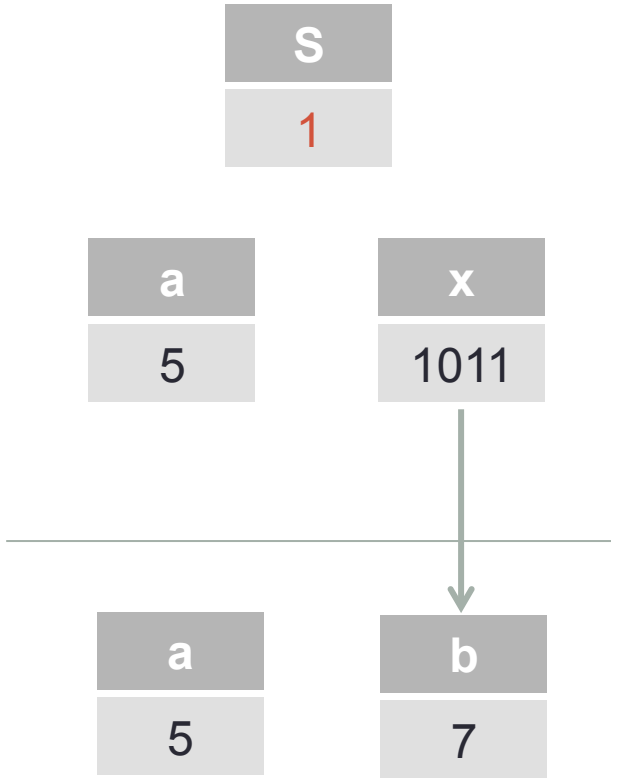
Ponteiros – exemplo 2

```c
#include <stdio.h>

void func(int a, int *x){
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main(){
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d",a,b);
}
```



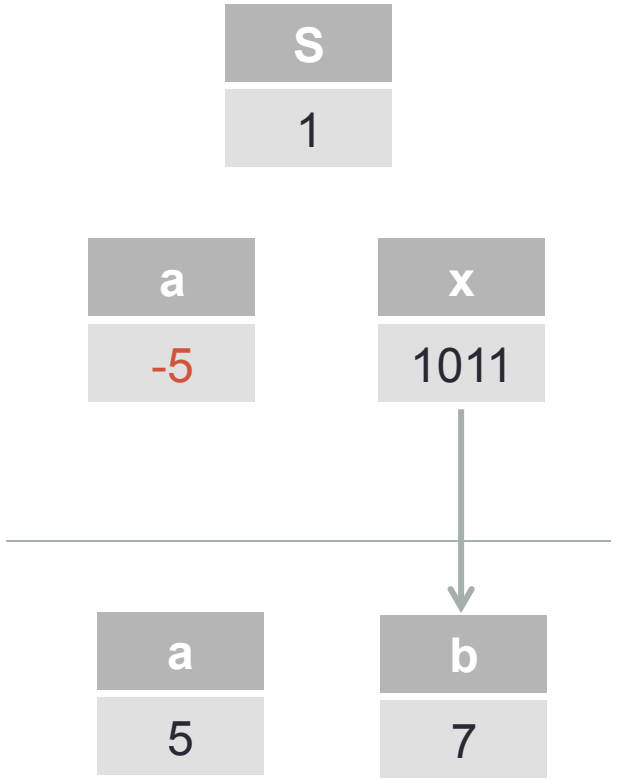
Ponteiros – exemplo 2

```c
#include <stdio.h>

void func(int a, int *x){
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main(){
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d",a,b);
}
```



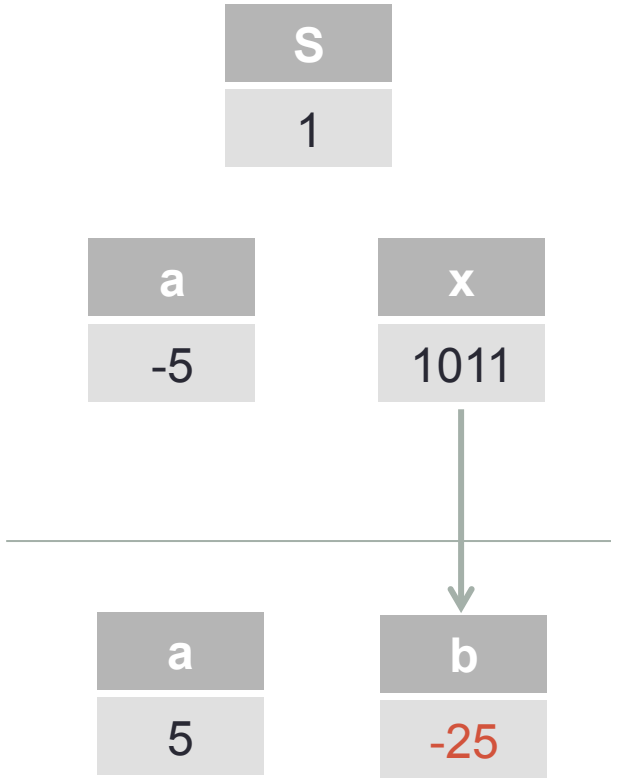
Ponteiros – exemplo 2

```c
#include <stdio.h>

void func(int a, int *x){
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main(){
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d",a,b);
}
```



Ponteiros – exemplo 2

```c
#include <stdio.h>

void func(int a, int *x){
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main(){
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d",a,b);
}
```



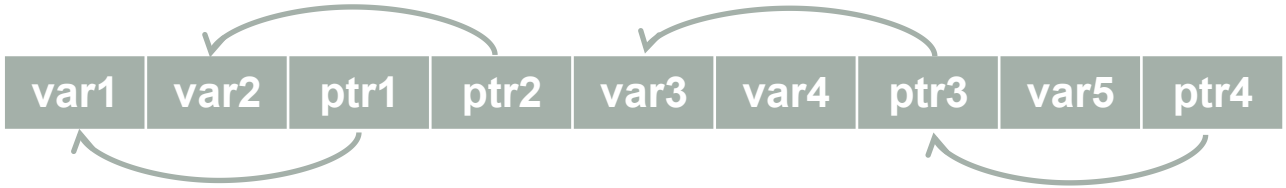
Ponteiros – exemplo 2

```c
#include <stdio.h>

void func(int a, int *x){
    int s = a%2;
    a = (1-2*s)*a;
    *x = a*5;
}

int main(){
    int a = 5, b = 7;
    func(a, &b);
    printf("%d-%d",a,b);
}
```

a	b
5	-25

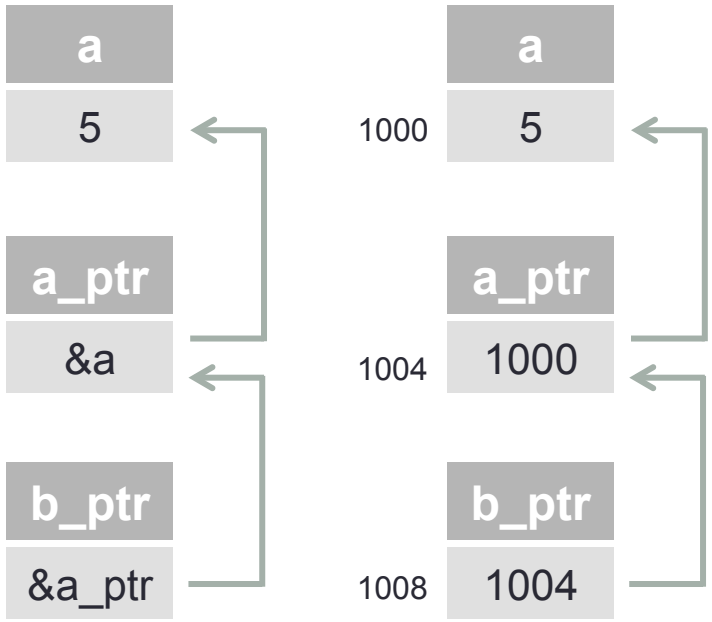
Ponteiros – em memória

- Na realidade os ponteiros são também variáveis
  - São guardados em memória em conjunto com as variáveis restantes
  - Têm um endereço associado
  - Podem ser "apontados" por outro apontador



Ponteiros para ponteiros

```c
#include <stdio.h>
int main()
{
  int a, *a_ptr, **b_ptr;

  a=5;
  a_ptr = &a;
  b_ptr= &a_ptr;

  printf("Val = %d", b_ptr);
  printf("Val = %d", *b_ptr);
  printf("Val = %d", **b_ptr);
}
```



Ponteiros para structs

O operador -> facilita o acesso às variáveis da struct – evita o uso de ()

```c
#include <stdio.h>

typedef struct {
        int a,b;
} ParV;

int main(){

  ParV pa;
  ParV *p2 = &pa;

  // acessar a struct
  (*p2).a = 3;
  (*p2).b = 5;
}
```

```c
#include <stdio.h>

typedef struct {
        int a,b;
} ParV;

int main(){

  ParV pa;
  ParV *p2 = &pa;

  // ou usar o operador ->
  p2->a = 3;
  p2->b = 5;
}
```

Vetores, ponteiros e Alocação Dinâmica

- Na linguagem C o nome de um vetor corresponde ao endereço do seu primeiro elemento, isto é, se v é um vetor v **==** **&**v**[**0**]**

- Existem 2 formas de colocar um ponteiro apontando para o primeiro elemento de um vetor:

```
int v[3] = {10,20,30};
int *ptr;

ptr = &v[0]; // primeira forma
ptr = v;     // segunda forma
```

Vetores, ponteiros e Alocação Dinâmica

- Também é possível apontar para outros lementos do vetor:

```
ptr = &v[2];
```

- **Este conceito é um dos dogmas centrais da manipulação de vetores através de ponteiros na linguagem C;**

- Usando a aritmética de ponteiros (será mostrado na sequência, é possível manipular os elementos de vetores e matrizes através de seus endereços de memória;

Aritmética de Ponteiros

- A aritmética de ponteiros nos permite realizar as operações de: incremento, decremento, diferença e comparação de ponteiros;
- Estas operações quando realizadas com os ponteiros são aplicadas ao endereço de memória que o ponteiro representa e são frequentemente utilizadas para manipulação de vetores;

```
int x = 5, *px = &x;
double y = 5, *py = &y;
printf("%p %ld\n", px, (long)px);
printf("%p %ld\n", px+1, (long)(px+1));
printf("%p %ld\n", py, (long)py);
printf("%p %ld\n", py+1, (long)(py+1));
```

Saída:
000000000022FE3C 2293308
000000000022FE40 2293312
000000000022FE30 2293296
000000000022FE38 2293304

- O exemplo acima mostra para os dois tipos de dados (int e float) o endereço do ponteiro na representação interna (flag %p do printf) e o mesmo valor convertido para long para efeito de análise.
- Observe que `px+1` incrementou em 4 bytes o ponteiro (int ocupa 4 bytes) e `py+1` incrementou em 8 bytes (double ocupa 8 bytes);

Aritmética de Ponteiros

- Incremento ou decremento - exemplos:

```
// incrementa ptr em 1: 1*(tipo de dado)
ptr++;
// incrementa ptr em 2: 2*(tipo de dado)
ptr = ptr + 2;
```

- Se ptr for um ponteiro para um int (4 bytes):

```
// incrementa ptr em 1: 1*(4)
ptr++;
// incrementa ptr em 2: 2*(4)
ptr = ptr + 2;
```

- Se ptr for um ponteiro para um double (8 bytes):

```
// incrementa ptr em 1: 1*(8)
ptr++;
// incrementa ptr em 2: 2*(8)
ptr = ptr + 2;
```

Aritmética de Ponteiros

Resumo das Operações sobre Ponteiros

Operação	Exemplo	Observações
Atribuição	ptr = &x	Podemos atribuir um valor (endereço) a um ponteiro. Se quisermos que aponte para nada podemos atribuir-lhe o valor da constante NULL.
Incremento	ptr=ptr+2	Incremento de 2*sizeof(tipo) de ptr.
Decremento	ptr=ptr-10	Decremento de 10*sizeof(tipo) de ptr.
Apontado por	*ptr	O operador asterisco permite obter o valor existente na posição cujo endereço está armazenado em ptr.
Endereço de	&ptr	Tal como qualquer outra variável, um ponteiro ocupa espaço em memória. Dessa forma podemos saber qual o endereço que um ponteiro ocupa em memória.
Diferença	ptr1 - ptr2	Permite-nos saber qual o nº de elementos entre ptr1 e ptr2.
Comparação	ptr1 > ptr2	Permite-nos verificar, por exemplo, qual a ordem de dois elementos num vetor através do valor dos seus endereços.