Algoritmos e estruturas de Dados

Ponteiros

Prof. Maiquel de Brito

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;
...
```

- Quando o ponteiro p é criado (linha 1), ele aponta para um local indefinido
- (pode até mesmo apontar para áreas reservadas ao sistema operacional)

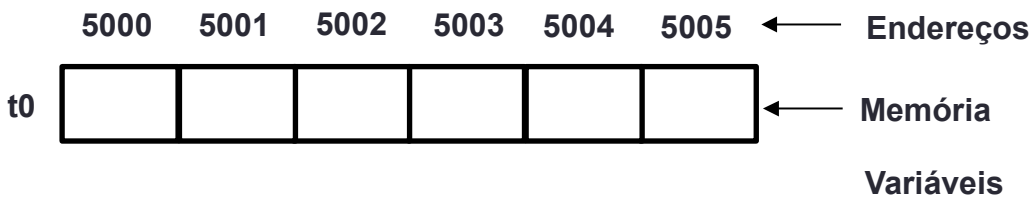
```
...
1 int *p;
2 int  a = 12;
3 p = &a;
...
```

- Devemos atribuir endereços de memória aos ponteiros
- mas como saber o endereço de memória de uma variável?

- O operador & retorna o endereço de memória de uma variável
- na linha 3, &a retorna o endereço da variável a e o atribui ao ponteiro p

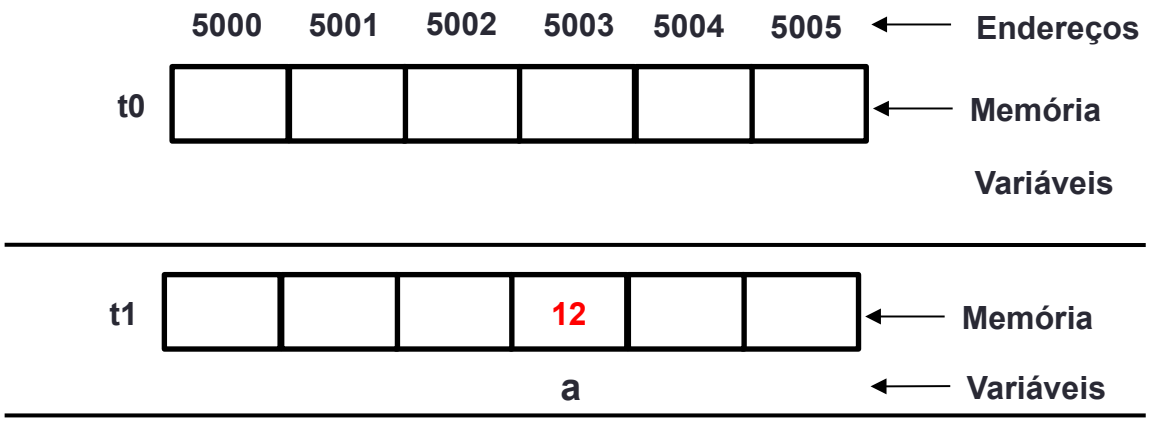
Ver exemplo nos próximos slides

Exemplo:
t0: nenhuma variável criada



Exemplo:
t1: criação da variável *a*

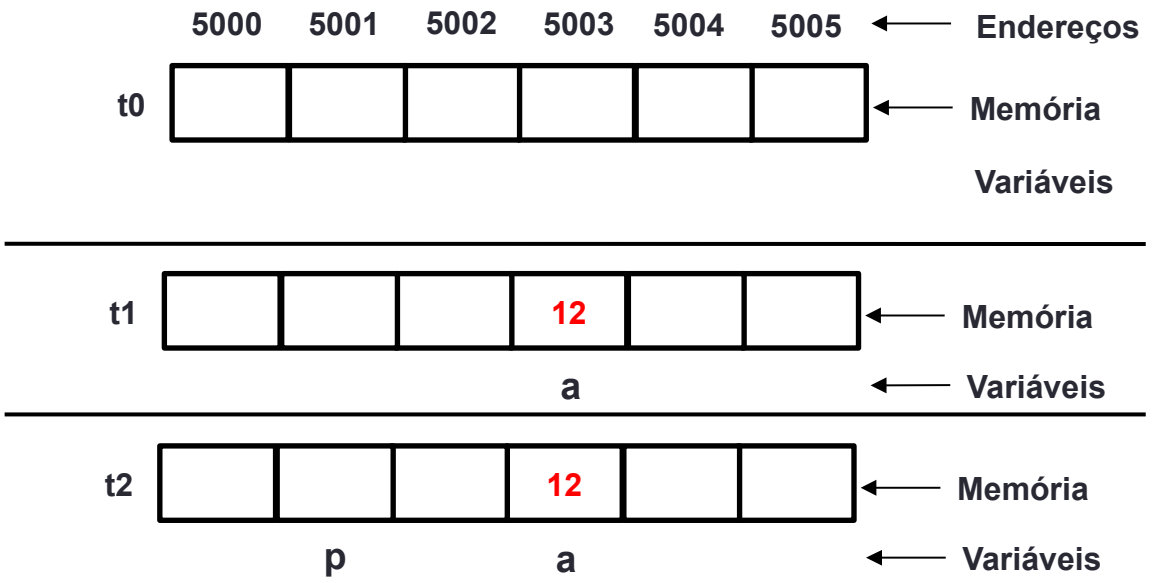
```
...
int   a = 12;   // t1



...
```



Exemplo:
t2: criação do ponteiro *p*

Nenhum valor é atribuído a *p* (ou seja, *p* não aponta a lugar algum).

```
...
int   a = 12;   // t1
int *p;         // t2


...
```

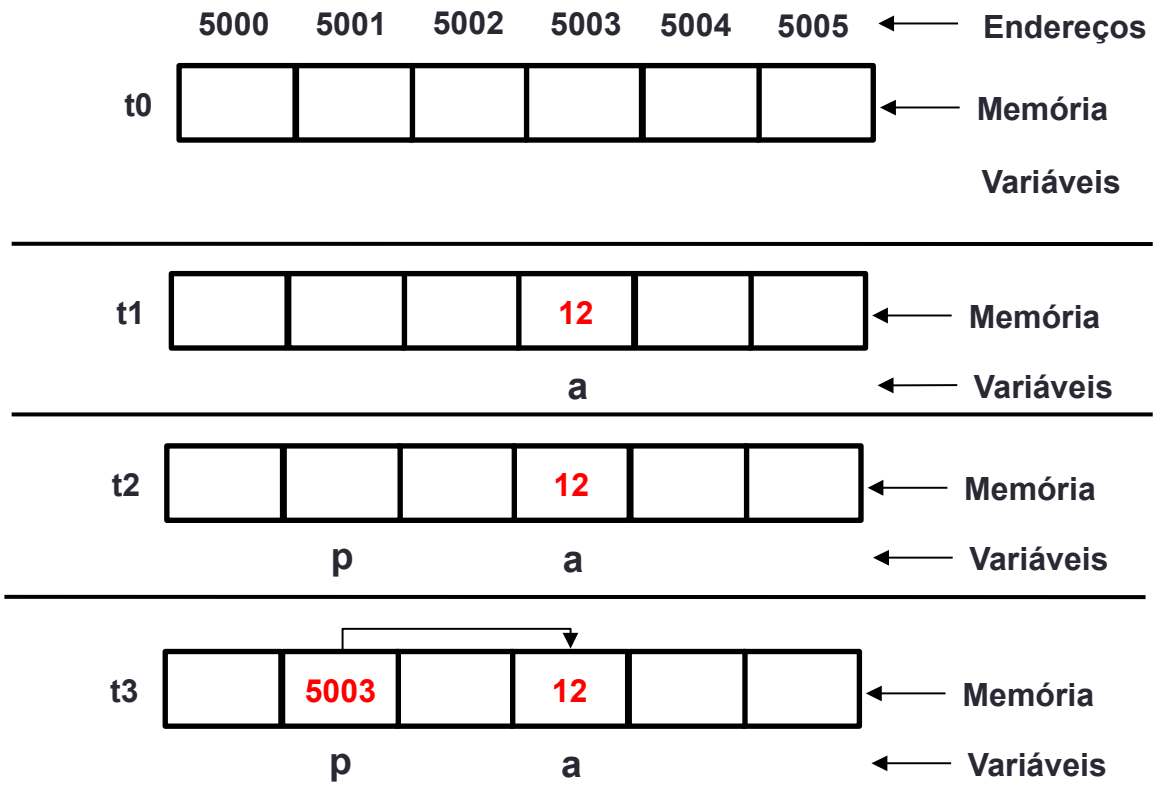


Exemplo:

t3: o endereço de *a* é atribuído a *p* (ou seja, *p* "aponta" para *a*)

Operador "&" retorna o endereço da variável que o sucede

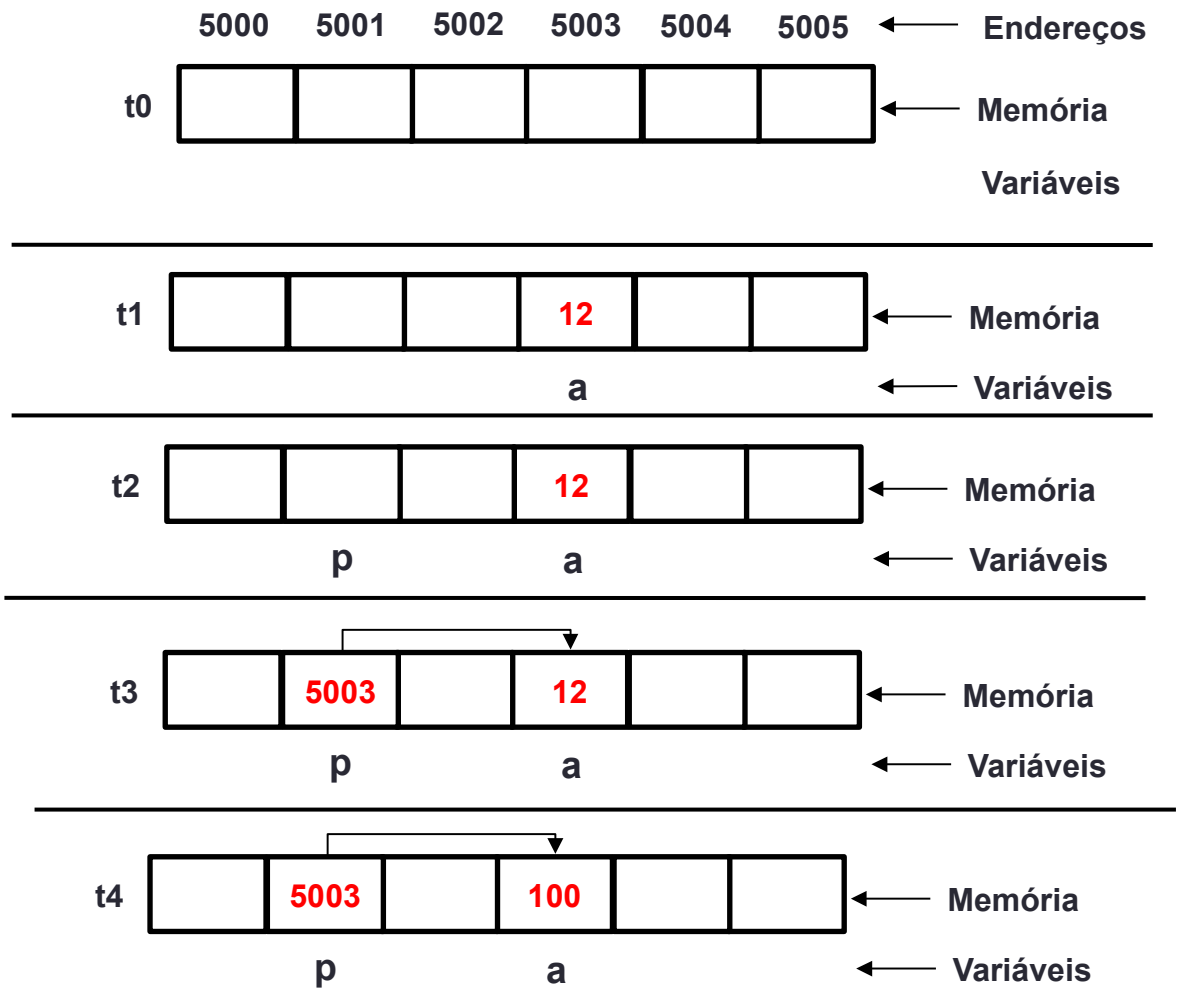
```
...
int   a = 12;   // t1
int *p;         // t2
p = &a;         // t3

...
```



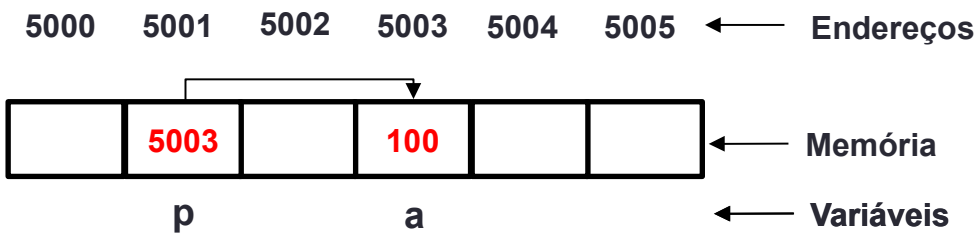
Exemplo:
t4: alteração do conteúdo do endereço para onde *p* aponta

Operador "*" - retorna o conteúdo do endereço para onde aponta o ponteiro que o sucede

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



Operadores úteis



`&var`: endereço de memória da variável *var*

ex.: &a → 5003

&p → 5001

`*ptr`: conteúdo do endereço de memória indicado pelo ponteiro *ptr*

*ex.: *p → 100*

```c
#include <stdio.h>
int main(){
  int a = 10;
  int *ptr = &a;
  printf("1. valor de a: %d\n",a);
  printf("2. endereco de memoria onde esta o valor de a: %p\n",&a);
  printf("3. endereco de memoria armazenado no ponteiro ptr: %p\n", ptr);
  printf("4. conteudo do endereco de memoria armazenado em ptr: %d\n",*ptr);
  printf("5. conteudo do endereco de memoria que armazena a variavel a:
                                                          %d\n",*(&a));
}
```

Exemplo de saída do programa acima:

```
1. valor de a: 10
2. endereco de memoria onde esta o valor de a: 000000000022FE44
3. endereco de memoria armazenado no ponteiro ptr: 000000000022FE44
4. conteudo do endereco de memoria armazenado em ptr: 10
5. conteudo do endereco de memoria armazenado da variavel a: 10
```

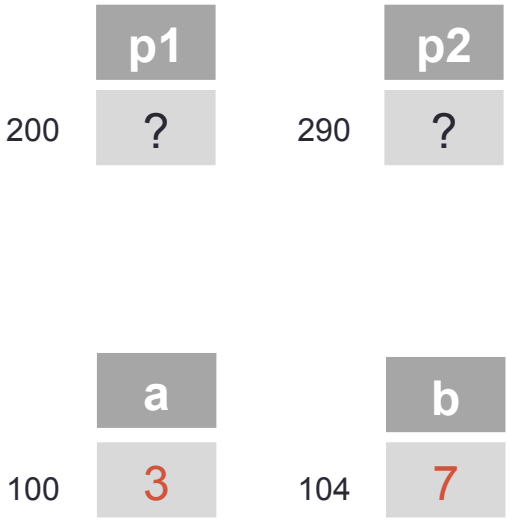
Ponteiros – exemplo 1

```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?

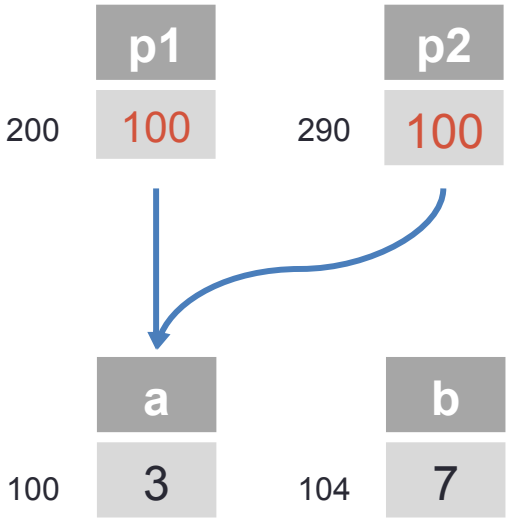
Ponteiros – exemplo 1

```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);
}
```



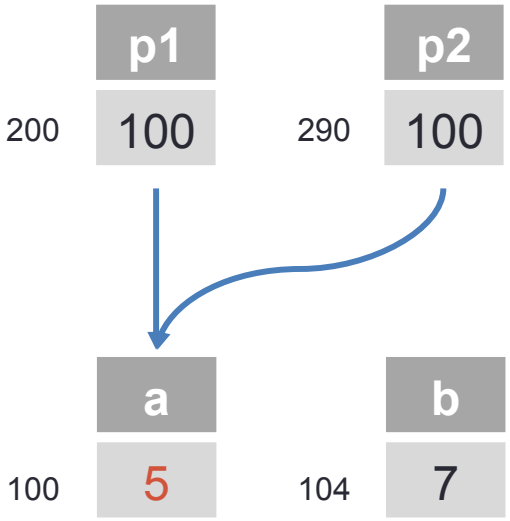
Ponteiros – exemplo 1

```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);
}
```



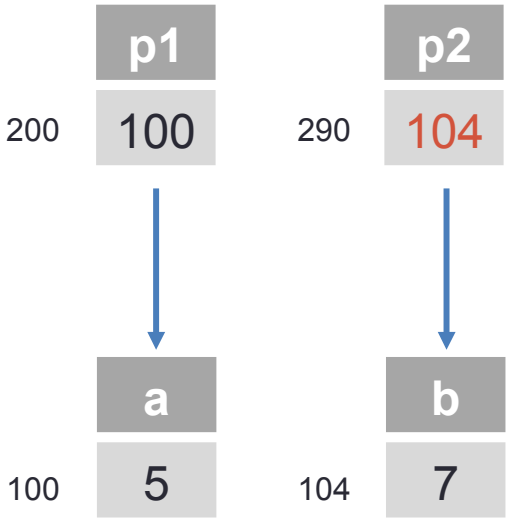
Ponteiros – exemplo 1

```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);
}
```

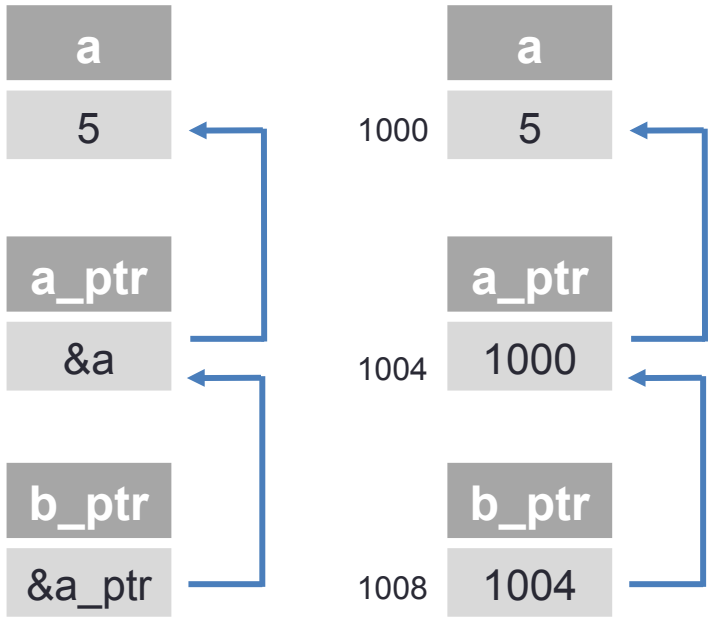


Ponteiros – em memória

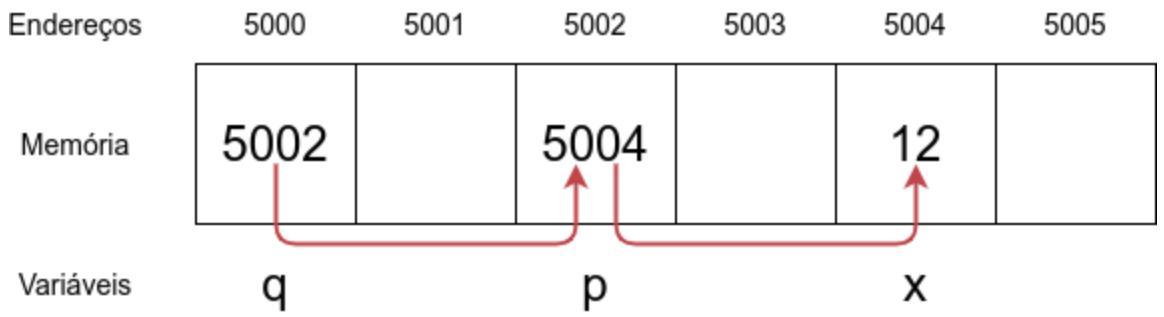
Na realidade os ponteiros são também variáveis
São guardados em memória em conjunto com as variáveis restantes
Têm um endereço associado
Podem ser "apontados" por outro apontador

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 ponteiros




```
int x = 12;
int *p = &x;
int **q = &p;
printf("%d", *(*q));
```

→ O programa imprimirá *12*

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)**

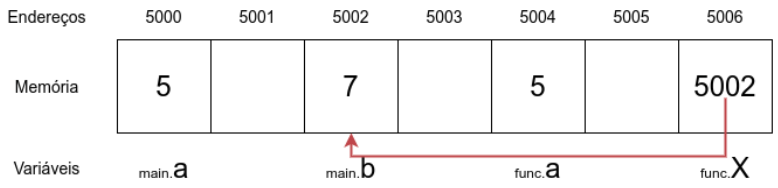
Saída:
Valor de a-b: 10 - 20
Valor de a-b: 20 - 10

Ponteiros como parâmetros

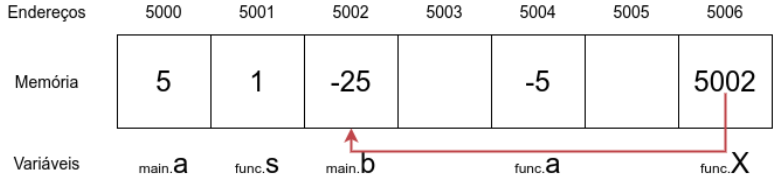
```c
#include <stdio.h>

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?

Início da execução da função *func:*



Final da execução da função *func:*



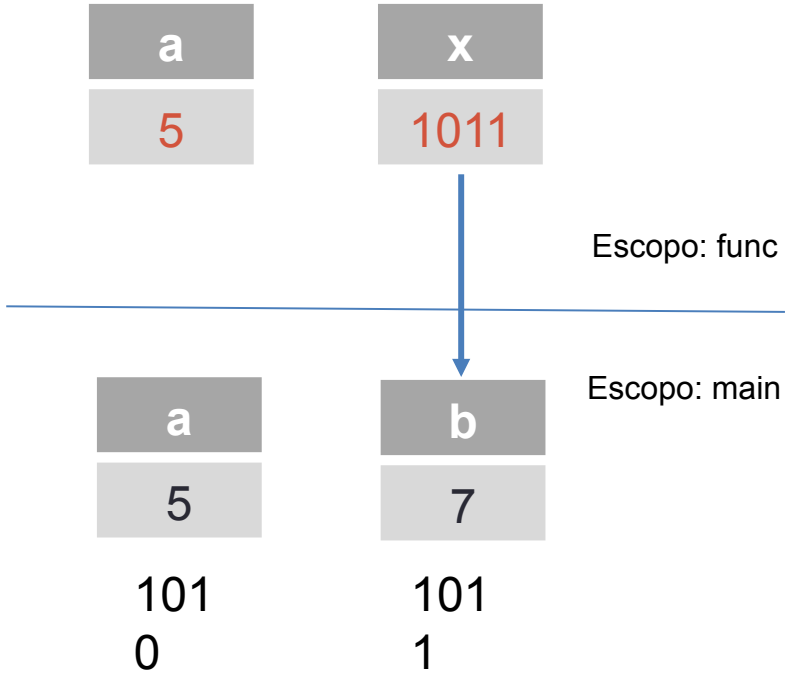
Ponteiros como parâmetros

```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
1010	1011

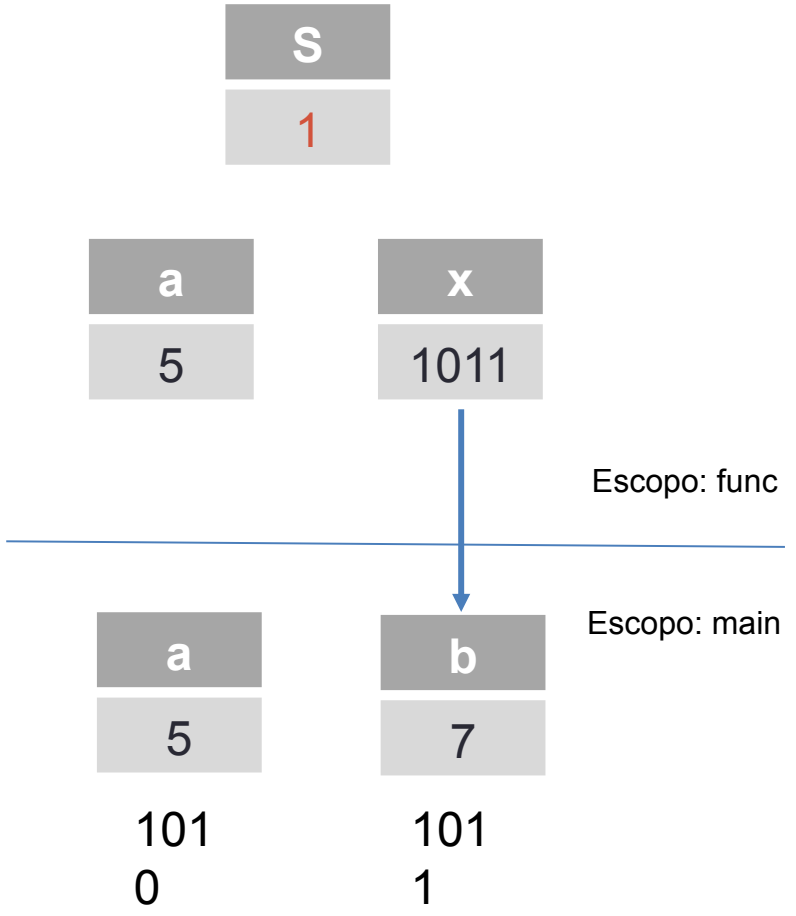
Ponteiros como parâmetros

```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);
}
```



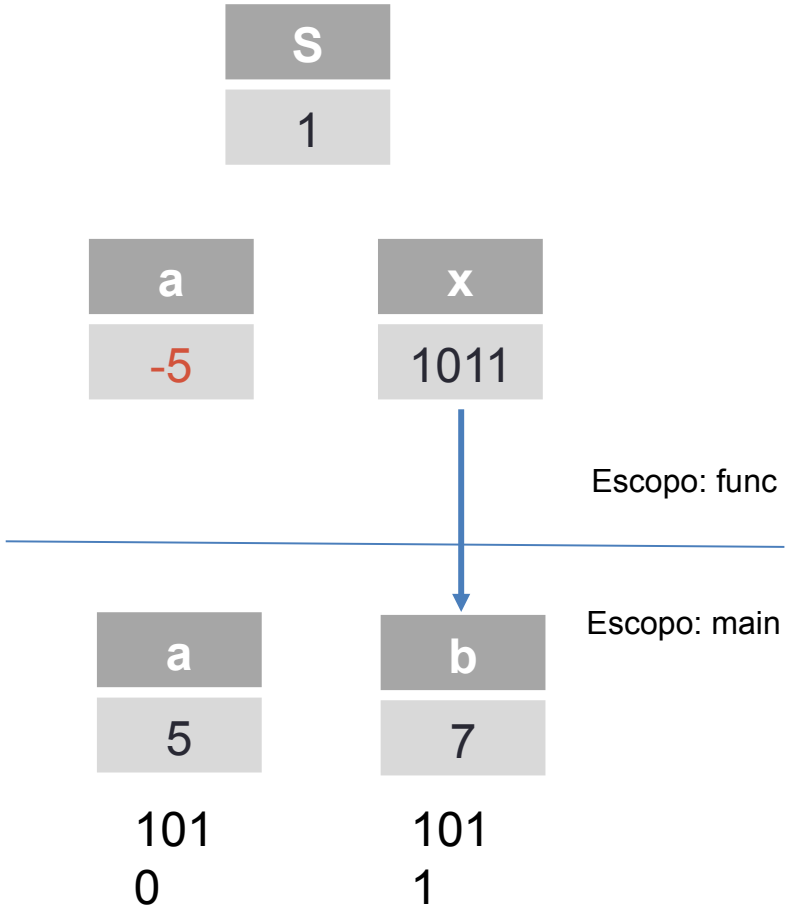
Ponteiros como parâmetros

```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);
}
```



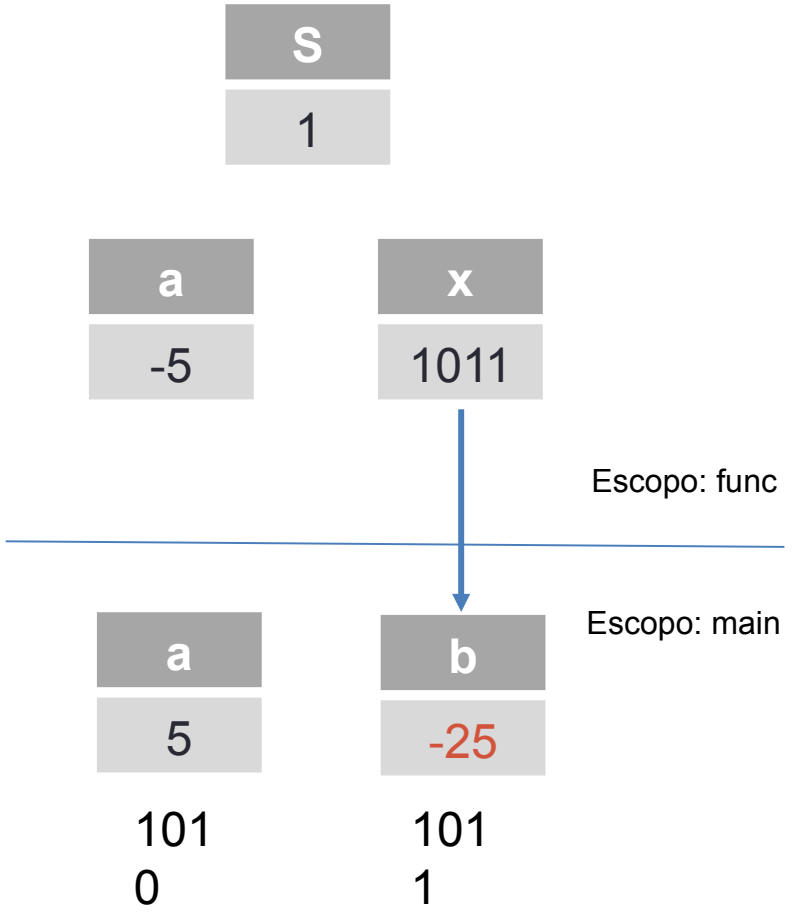
Ponteiros como parâmetros

```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);
}
```



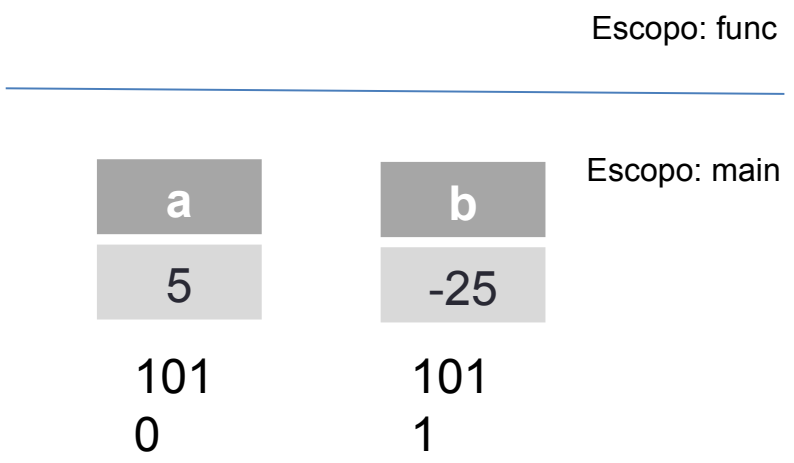
Ponteiros como parâmetros

```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 como parâmetros

```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);
}
```

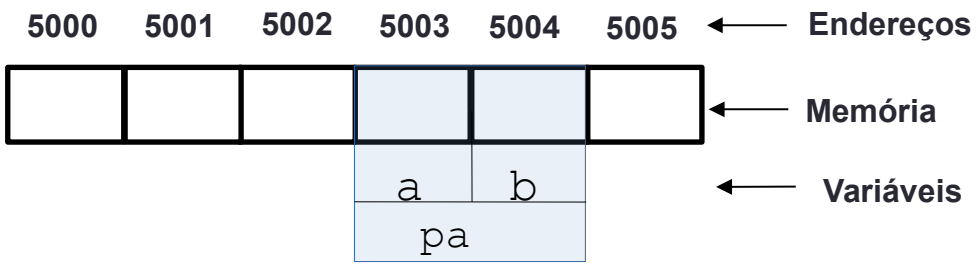


Ponteiros para structs

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

```
typedef struct {
    int a,b;
} ParV;

int main(){

  ParV pa;



}
```

```
typedef struct {
    int a,b;
} ParV;

int main(){

  ParV pa;



}
```

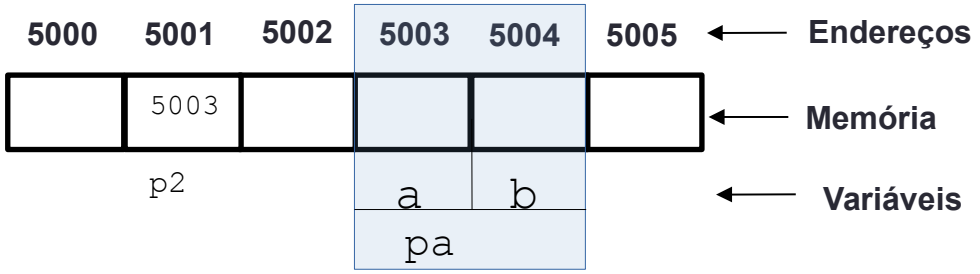


Ponteiros para structs

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

```
typedef struct {
    int a,b;
} ParV;

int main(){

  ParV pa;
  ParV *p2 = &pa;



}
```

```
typedef struct {
    int a,b;
} ParV;

int main(){

  ParV pa;
  ParV *p2 = &pa;



}
```

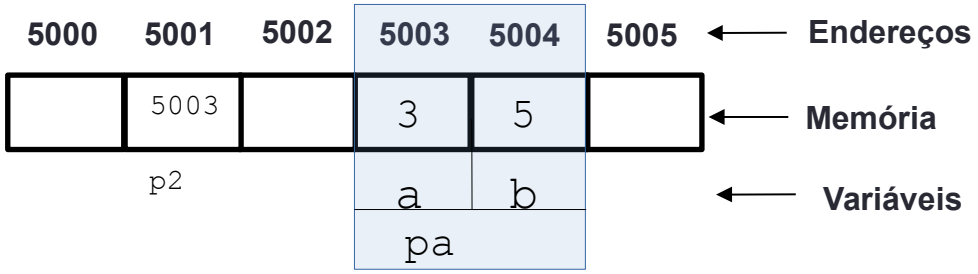


Ponteiros para structs

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

```c
typedef struct {
    int a,b;
} ParV;

int main(){

  ParV pa;
  ParV *p2 = &pa;

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

```c
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 elementos 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;

```
int v[5] = {11,22,33,44,55};
int *p = v;

printf("%d\n",*p+2);
printf("%d\n",*(p+2));
```

Aritmética de Ponteiros

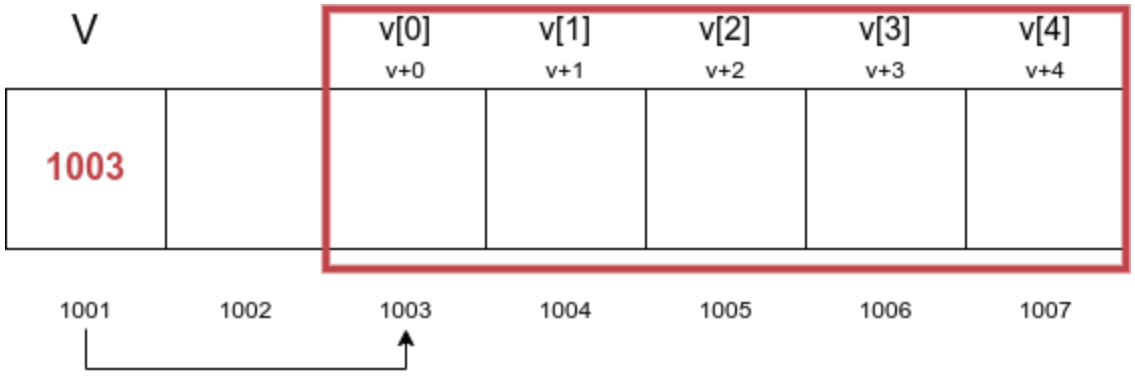
Mais informações sobre vetores e aritmética de ponteiros podem ser encontradas no vídeo disponível em https://youtu.be/MBQQR82TQsQ

Exemplo - Aritmética de Ponteiros (slide 1/4)

```
int v[5];
```

Ao declarar um vetor *v* um de tamanho *s* de determinado *tipo* (no exemplo, `int`), cria-se um ponteiro para um bloco de memória com tamanho necessário para armazenar *s* variáveis daquele *tipo*.

O ponteiro aponta para o primeiro endereço daquele bloco de memória (no exemplo, o endereço é 1003).

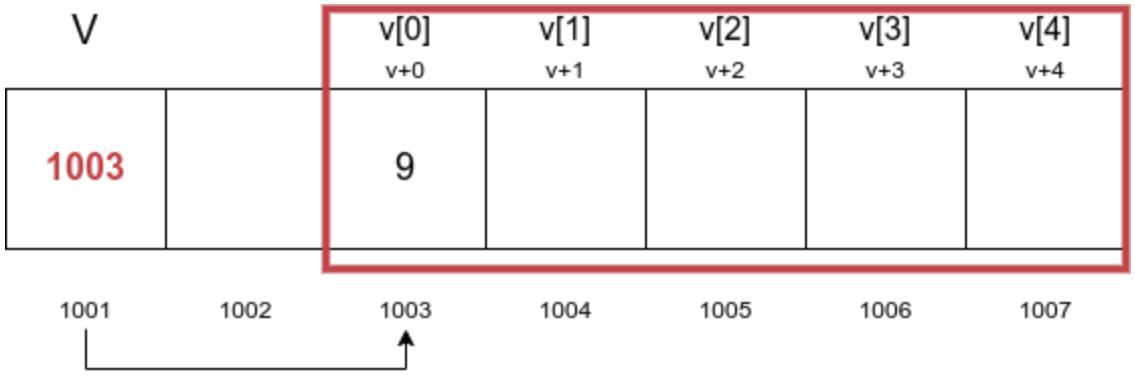


Exemplo - Aritmética de Ponteiros (slide 2/4)

```
int v[5];
v[0] = 9;
```

```
int v[5];
*(v+0) = 9;
```

Para acessar a i-*ésima* posição de um vetor, pode-se utilizar índices (no caso, `v[i]`) ou aritmética de ponteiros (no caso, `*(v+i)`). Tanto `v[i]` quanto `*(v+i)` expressam que se está acessando o *conteúdo armazenado no i-ésimo espaço consecutivo ao endereço inicial do bloco de memória que armazena o vetor* (ou seja, do endereço para onde *v* aponta).



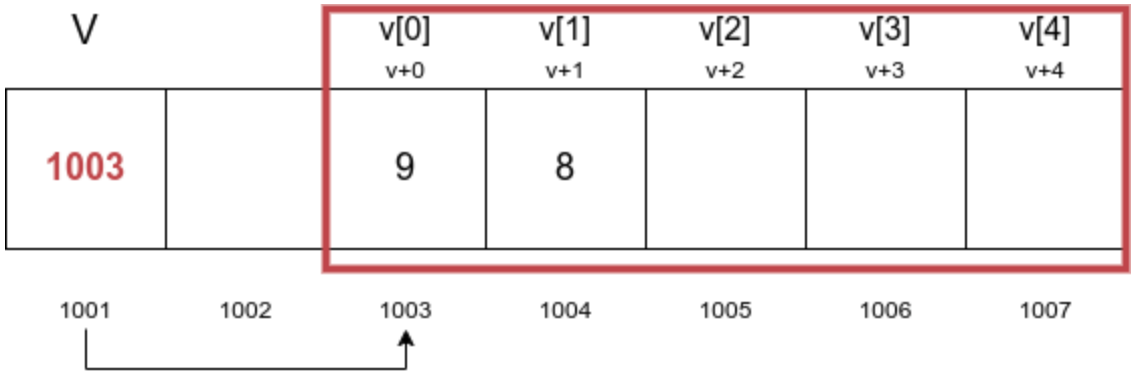
Exemplo - Aritmética de Ponteiros (slide 3/4)

```
int v[5];
v[0] = 9;
v[1] = 8;
```

```
int v[5];
*(v+0) = 9;
*(v+1) = 8;
```

Para acessar a i-*ésima* posição de um vetor, pode-se utilizar índices (no caso, `v[i]`) ou aritmética de ponteiros (no caso, `*(v+i)`). Tanto `v[i]` quanto `*(v+i)` expressam que se está acessando o *conteúdo armazenado no i-ésimo espaço consecutivo ao endereço inicial do bloco de memória que armazena o vetor* (ou seja, do endereço para onde *v* aponta).

No exemplo, tanto `v[1]` quanto `(v+1)` apontam para 1 espaço após o início do vetor.

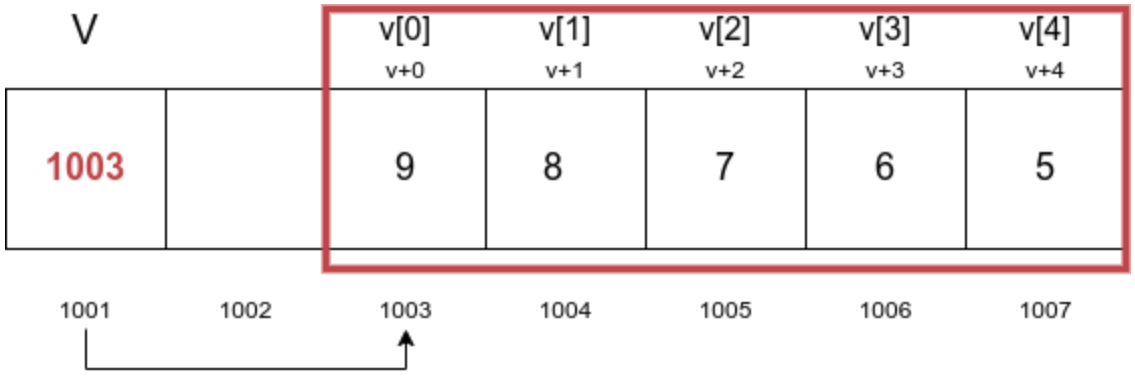


Exemplo - Aritmética de Ponteiros (slide 4/4)

```
int v[5];
v[0] = 9;
v[1] = 8;
v[2] = 7;
v[3] = 6;
v[4] = 5;
```

```
int v[5];
*(v+0) = 9;
*(v+1) = 8;
*(v+2) = 7;
*(v+3) = 6;
*(v+4) = 5;
```

Os programas da esquerda e da direita têm o mesmo comportamento.



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.

Ponteiros, vetores e *scanf*

O segundo parâmetro do *scanf* é um endereço de memória (ex. *scanf("%d", &i);*
O *scanf* lê valores e salva-os no endereço indicado pelo segundo parâmetro

Ao tratar com vetores, v[i] = *(v+i).
Para acessar o endereço de v[i], deve-se usar `&v[i]` ou `(v+i)`.

```
int i, v[3];
for(i=0;i<3;i++)
   scanf("%d",v[i]);
```

Gera erro pois v[i] não é um endereço

```
int i, v[3];
for(i=0;i<3;i++)
  scanf("%d",v);
```

Salva todos os valores na primeira posição
porque v é o endereço do 1º elemento

```
int i, v[3];
for(i=0;i<3;i++)
  scanf("%d",&v[i]);
```

ou

```
int i, v[3];
for(i=0;i<3;i++)
  scanf("%d",(v+i));
```

Lê corretamente

Ponteiros para funções

Funções também são armazenadas em algum local da memória (que possui um endereço

Pode-se usar ponteiros para acessar as funções pelos seus endereços em vez de acessá-las pelos seus nomes.

Declaração de um ponteiro para função:

```
tipo_de_retorno (*nome_do_ponteiro)(parâmetros)
```

ou

```
tipo_de_retorno (*nome_do_ponteiro)()
```
(no caso de função sem parâmetros)

Ponteiros para funções

Funções também são armazenadas em algum local da memória (que possui um endereço

Pode-se usar ponteiros para acessar as funções pelos seus endereços em vez de acessá-las pelos seus nomes.

```
int soma(int x, int y){
    return x + y;
}

int produto(int x, int y){
    return x * y;
}

int main(){
    int (*p)(int a, int b);
    p = soma;
    printf("%d\n", p(2,3));
    p = produto;
    printf("%d\n", p(2,3));
}
```

Ponteiros para funções como parâmetros

Funções podem ser passadas como parâmetros para funções através de ponteiros para funções:

```
int soma(int x, int y){
   return x + y;
}

int produto(int x, int y){
   return x * y;
}

void executa(int (*f)(int, int), int a, int b){
   printf("Resultado: %d\n", f(2,3));
}

int main(){
   executa(soma,2,3);
   executa(produto,2,3);
}
```