

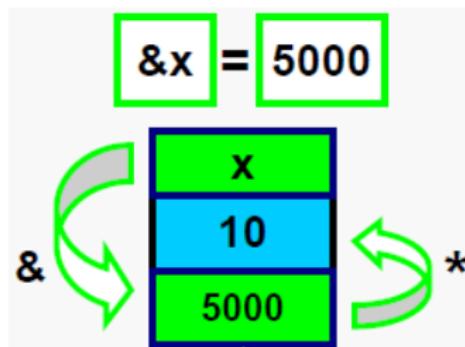
BCC 201 - Introdução à Programação I

Ponteiros

Guillermo Cámara-Chávez
UFOP

Ponteiros ... I

```
int main()
{
    int x;
    x = 10
    printf("Conteudo de x: %d \n", x);
    printf("Endereco de x: %p \n", &x);
    return 0;
}
```



Ponteiros ... II

0x10	?
0x11	?
0x12	?
0x13	?
0x14	?
0x15	?
0x16	?
0x17	?
0x18	?
0x19	?
0x20	?
0x21	?
0x22	?
0x23	?
0x24	?
0x25	?

```
int x, *px;
```

```
x = 23;
```

```
px = &x;
```

```
*px = 19;
```

Ponteiros ... III

0x10	?
0x11	
0x12	?
0x13	
0x14	
0x15	?
0x16	?
0x17	?
0x18	
0x19	?
0x20	
0x21	
0x22	?
0x23	?
0x24	?
0x25	?

x → int x, *px;
x = 23;
px = &x;
*px = 19;

px

Ponteiros ... IV

0x10	?
0x11	
0x12	23
0x13	
0x14	
0x15	?
0x16	?
0x17	?
0x18	
0x19	?
0x20	
0x21	
0x22	?
0x23	?
0x24	?
0x25	?

x

px

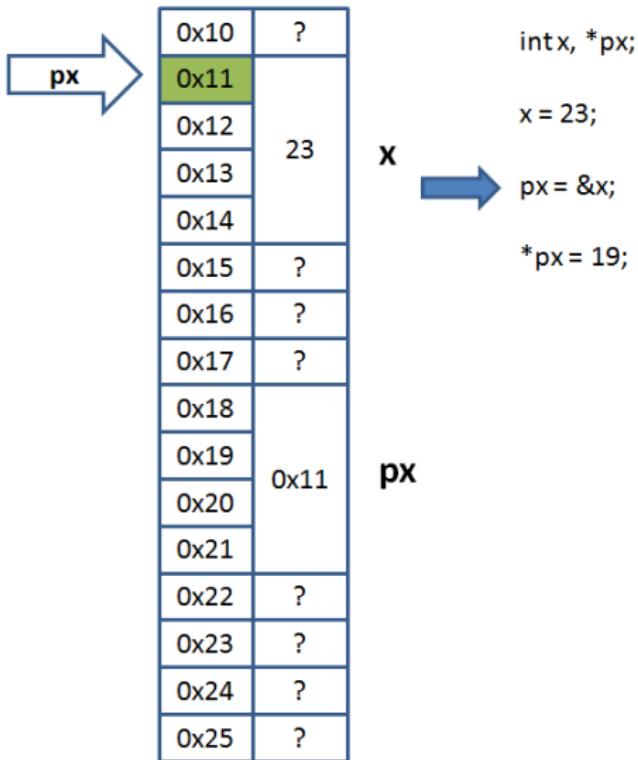
int x, *px;

x = 23;

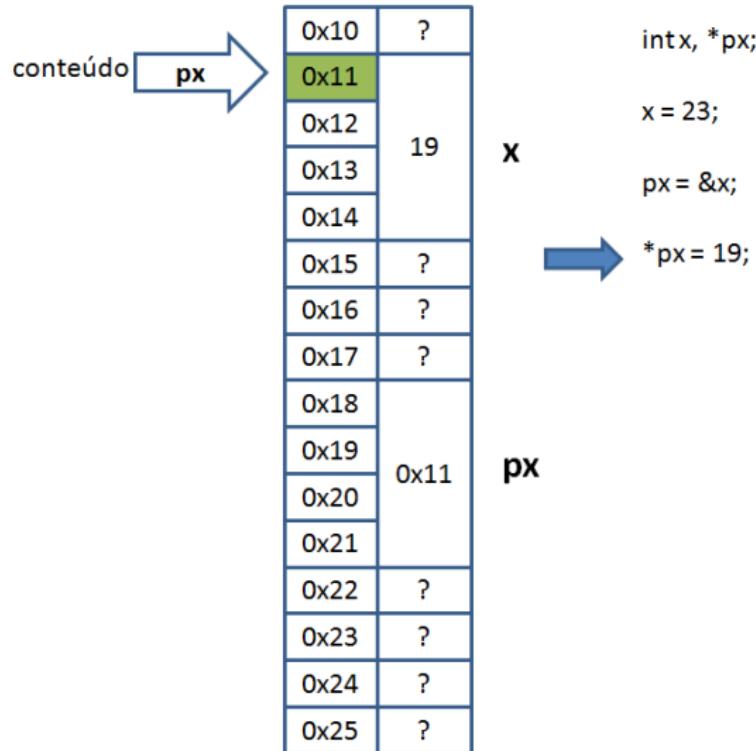
px = &x;

*px = 19;

Ponteiros ... V



Ponteiros ... VI



Ponteiros ... VII

0x12	?
0x16	?
0x20	?
0x24	?
0x28	?
0x32	?
0x36	?
0x40	?
0x44	?
0x48	?
0x52	?
0x56	?
0x60	?
0x64	?
0x68	?
0x72	?

```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... VIII

0x12	?
0x16	?
0x20	?
0x24	?
0x28	?
0x32	?
0x36	?
0x40	?
0x44	?
0x48	?
0x52	?
0x56	?
0x60	100
0x64	200
0x68	?
0x72	?

x } main
y }

```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... IX

0x12	?
0x16	?
0x20	?
0x24	?
0x28	?
0x32	?
0x36	?
0x40	?
0x44	?
0x48	?
0x52	?
0x56	?
0x60	100
0x64	200
0x68	?
0x72	?

x } main
y }

```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... X

0x12	?
0x16	?
0x20	?
0x24	100
0x28	200
0x32	?
0x36	?
0x40	?
0x44	?
0x48	?
0x52	?
0x56	?
0x60	100
0x64	200
0x68	?
0x72	?

x
y
aux

Nao_troca

x
y

main

```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... XI

0x12	?
0x16	?
0x20	?
0x24	100
0x28	200
0x32	100
0x36	?
0x40	?
0x44	?
0x48	?
0x52	?
0x56	?
0x60	100
0x64	200
0x68	?
0x72	?

x
y
aux

Nao_troca

x
y

main



```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... XII

0x12	?
0x16	?
0x20	?
0x24	200
0x28	200
0x32	100
0x36	?
0x40	?
0x44	?
0x48	?
0x52	?
0x56	?
0x60	100
0x64	200
0x68	?
0x72	?



```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... XIII

0x12	?
0x16	?
0x20	?
0x24	200
0x28	100
0x32	100
0x36	?
0x40	?
0x44	?
0x48	?
0x52	?
0x56	?
0x60	100
0x64	200
0x68	?
0x72	?

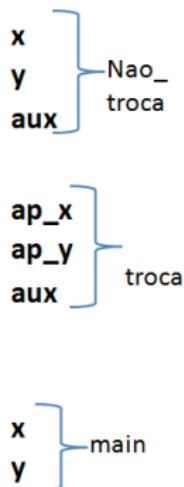
x } Nao_troca
y }
aux }

x } main
y }

```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... XIV

0x12	?
0x16	?
0x20	?
0x24	200
0x28	100
0x32	100
0x36	?
0x40	?
0x44	?
0x48	?
0x52	?
0x56	?
0x60	100
0x64	200
0x68	?
0x72	?



```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... XV

0x12	?
0x16	?
0x20	?
0x24	200
0x28	100
0x32	100
0x36	?
0x40	0x60
0x44	0x64
0x48	?
0x52	?
0x56	?
0x60	100
0x64	200
0x68	?
0x72	?

Diagram illustrating pointer assignments:

- Two blue arrows point from the labels `ap_x` and `ap_y` to the memory locations `0x60` and `0x64` respectively.
- The label `x` is associated with the value `100` at address `0x60`.
- The label `y` is associated with the value `200` at address `0x64`.
- The label `aux` is associated with the value `?` at address `0x68`.
- The label `ap_x` is associated with the value `0x60` at address `0x40`.
- The label `ap_y` is associated with the value `0x64` at address `0x44`.
- The label `aux` is associated with the value `?` at address `0x48`.
- The label `x` is associated with the value `?` at address `0x52`.
- The label `y` is associated with the value `?` at address `0x56`.
- The label `main` is associated with the values `100` and `200` at addresses `0x60` and `0x64` respectively.

```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... XVI

0x12	?
0x16	?
0x20	?
0x24	200
0x28	100
0x32	100
0x36	?
0x40	0x60
0x44	0x64
0x48	100
0x52	?
0x56	?
0x60	100
0x64	200
0x68	?
0x72	?

Diagram illustrating pointer assignments:

- Two arrows point to the first two columns of the table:
 - The top arrow is labeled `ap_x`.
 - The bottom arrow is labeled `ap_y`.
- Braces indicate variable associations:
 - `x`, `y`, and `aux` are grouped under the label `Nao_troca`.
 - `ap_x`, `ap_y`, and `aux` are grouped under the label `troca`.
 - `x` and `y` are grouped under the label `main`.

```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... XVII

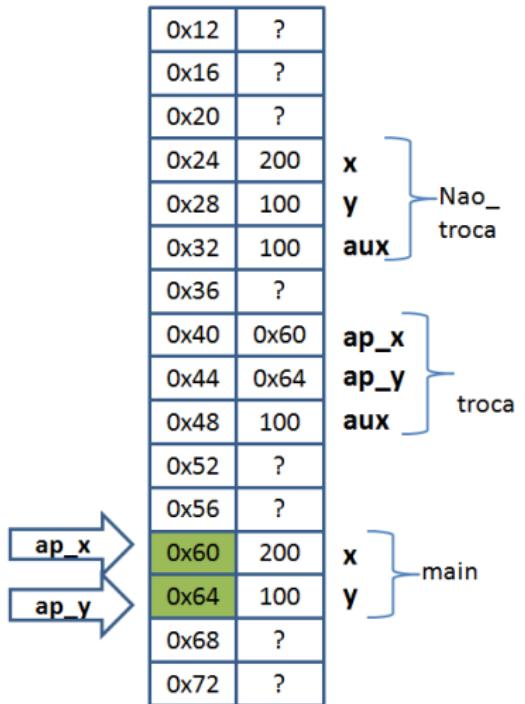
0x12	?
0x16	?
0x20	?
0x24	200
0x28	100
0x32	100
0x36	?
0x40	0x60
0x44	0x64
0x48	100
0x52	?
0x56	?
0x60	200
0x64	200
0x68	?
0x72	?

Diagram illustrating pointer assignments:

- Two blue arrows point from the bottom of the table to the memory locations 0x60 and 0x64.
- The label **ap_x** is associated with the value at 0x60 (200).
- The label **ap_y** is associated with the value at 0x64 (200).
- Brackets group variables:
 - x**, **y**, and **aux** are grouped under the label **Nao_troca**.
 - ap_x**, **ap_y**, and **aux** are grouped under the label **troca**.
 - x** and **y** are grouped under the label **main**.

```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Ponteiros ... XVIII

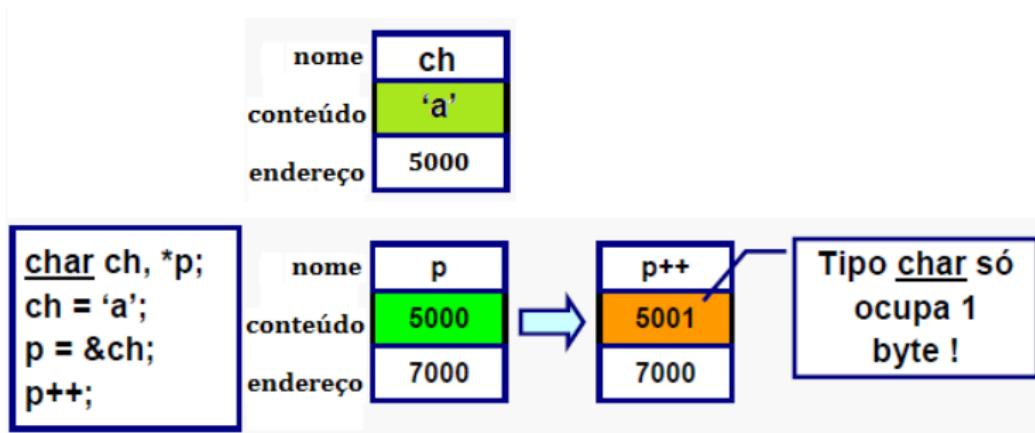


```
void troca(int *ap_x, int *ap_y) {  
    int aux;  
    aux = *ap_x;  
    *ap_x = *ap_y;  
    *ap_y = aux;  
}  
void nao_troca(int x, int y) {  
    int aux;  
    aux = x;  
    x = y;  
    y = aux;  
}  
int main() {  
    int x = 100, y = 200;  
    nao_troca(x, y);  
    printf("x = %d y = %d \n", x, y);  
    troca(&x, &y);  
    printf("x = %d y = %d \n", x, y);  
    return 0;  
}
```

Aritmética de Ponteiros I

- ▶ Uma variável do tipo ponteiro está sempre associada a um tipo
- ▶ Um ponteiro para um dado tipo t endereça o número de bytes que esse tipo t ocupa na memória, i.e., endereça **sizeof(t)** bytes.
- ▶ Se um ponteiro para uma variável do tipo t for incrementada através do operador `++`, automaticamente este ponteiro passará a ter o valor $x + \text{sizeof}(t)$

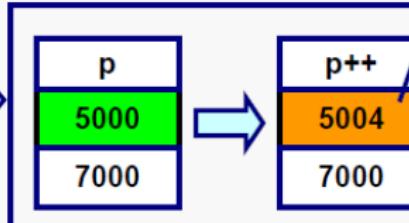
Aritmética de Ponteiros II



Aritmética de Ponteiros III

```
int x, *p;  
x = 10;  
p = &x;  
p++;
```

Memória



Tipo `int`
ocupa 4
bytes !

Em Detalhes

Espaço de 4 bytes para
armazenar a variável x tipo int.

A operação `p++` percorre
sizeof(tipo p) bytes !

p					p++			
x	x	x	x		?	?	?	?
5000	5001	5002	5003		5004	5005	5006	5007

Aritmética de Ponteiros IV

Endereço	Conteúdo	Nome
0x1000		
0x1001	a	letra
0x1002		
0x1003		
0x1004		
0x1005		
0x1006		
0x1007		
0x1008		
0x1009		
0x1010		
0x1011		
0x1012		
0x1013		
0x1014		
0x1015		
0x1016		

pLetra

```
int main() {
    char* pChar, letra = 'a';
    pLetra = &letra;
    ...
    pLetra++;
    return 0;
}
```

Aritmética de Ponteiros V

Endereço	Conteúdo	Nome
0x1000		
0x1001	a	letra
0x1002		
0x1003		
0x1004		
0x1005		
0x1006		
0x1007		
0x1008		
0x1009		
0x1010		
0x1011		
0x1012		
0x1013		
0x1014		
0x1015		
0x1016		

pLetra →

```
int main() {
    char* pChar, letra = 'a';
    pLetra = &letra;
    ...
    pLetra++;
    pLetra++;
    return 0;
}
```

Aritmética de Ponteiros VI

Endereço	Conteúdo	Nome
0x1000		
0x1001	a	letra
0x1002		
0x1003		
0x1004		
0x1005		
0x1006		
0x1007		
0x1008		
0x1009		
0x1010		
0x1011		
0x1012		
0x1013		
0x1014		
0x1015		
0x1016		

pLetra

→

```
int main() {  
    char* pChar, letra = 'a';  
    pLetra = &letra;  
    ...  
    pLetra++;  
    pLetra++;  
    return 0;  
}
```

Aritmética de Ponteiros VII

Endereço	Conteúdo	Nome
0x1000		
0x1001		
0x1002		
0x1003		
0x1004		
0x1005		
0x1006		
0x1007		
0x1008		
0x1009		
0x1010		
0x1011		
0x1012		
0x1013		
0x1014		
0x1015		
0x1016		

pNum

Endereço	Conteúdo	Nome
0x1000		
0x1001		
0x1002		
0x1003		
0x1004		
0x1005		
0x1006		
0x1007		
0x1008		
0x1009		
0x1010		
0x1011		
0x1012		
0x1013		
0x1014		
0x1015		
0x1016		

```
int main() {
    int* pNum, num = 20;
    pNum = &Num;
    ...
    pNum++;
    return 0;
}
```

Aritmética de Ponteiros VIII

Endereço	Conteúdo	Nome
0x1000		
0x1001		
0x1002		
0x1003		
0x1004		
0x1005	20	num
0x1006		
0x1007		
0x1008		
0x1009		
0x1010		
0x1011		
0x1012		
0x1013		
0x1014		
0x1015		
0x1016		

pNum →

```
int main() {
    int* pNum, num = 20;
    pNum = &Num;
    ...
    pNum++;
    return 0;
}
```

Aritmética de Ponteiros IX

Endereço	Conteúdo	Nome
0x1000		
0x1001		
0x1002		
0x1003		
0x1004		
0x1005		
0x1006		
0x1007		
0x1008		
0x1009		
0x1010		
0x1011		
0x1012		
0x1013		
0x1014		
0x1015		
0x1016		

pNum →

```
int main() {
    int* pNum, num = 20;
    pNum = &Num;
    ...
    pNum++;
    pNum++;

    return 0;
}
```

Aritmética de Ponteiros X

Endereço	Conteúdo	Nome
0x1000		
0x1001		
0x1002		
0x1003		
0x1004	20.0	
0x1005		num
0x1006		
0x1007		
0x1008		
0x1009		
0x1010		
0x1011		
0x1012		
0x1013		
0x1014		
0x1015		
0x1016		

pNum

```
int main() {
    double* pNum, num = 20.0;
    pNum = &Num;
    ...
    pNum++;
    return 0;
}
```

Aritmética de Ponteiros XI

Endereço	Conteúdo	Nome
0x1000		
0x1001		
0x1002		
0x1003		
0x1004	20.0	num
0x1005		
0x1006		
0x1007		
0x1008		
0x1009		
0x1010		
0x1011		
0x1012		
0x1013		
0x1014		
0x1015		
0x1016		

pNum →

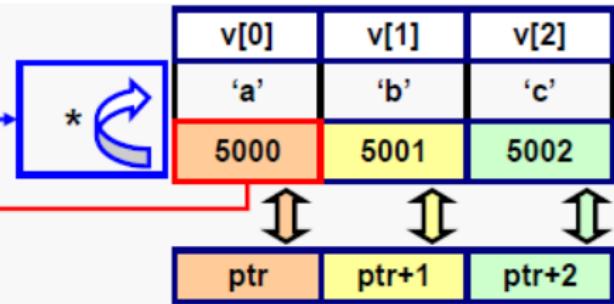
```
int main() {
    double* pNum, num = 20.0;
    pNum = &Num;
    ...
    pNum++;
    return 0;
}
```

Relação entre ponteiros e vetores I

- ▶ A aritmética de ponteiros é particularmente importante para manipulação de vetores e strings.
- ▶ Quando declaramos um vetor seus elementos são alocados em espaços de memória vizinhos.
- ▶ O nome de um vetor equivale ao endereço do primeiro elemento dele (se um vetor possui nome v , então, v equivale a $v[0]$).

Relação entre ponteiros e vetores II

```
main()
{ char *ptr, v[3]={'a', 'b', 'c'};
  ptr =v;
  printf("v[0] = %c", *(ptr));
  printf("v[1] = %c", *(++ptr));
  printf("v[2] = %c", *(++ptr));
}
```

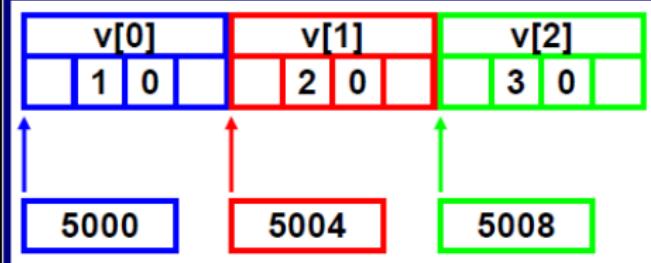


Vetores I

Exemplo – Vetor+Ponteiro

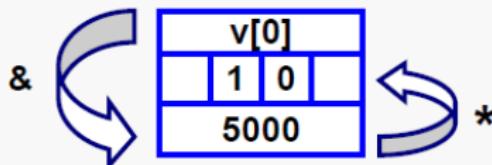
```
#include <stdio.h>
main()
{
    int i, *p, v[3];
    p = &v[0]; // ou p = &v;
    // Usando indices para acessar v.
    printf("Usando v[i]:");
    for (i=0; i < 3; i++)
    {
        scanf("%d", &v[i]);
        printf(" [%4d]", v[i]);
    }
    // Usando ponteiros.
    printf("Usando v[i]:");
    for (i=0; i < 3; i++, ptr++)
    {
        scanf("%d", ptr);
        printf(" [%4d]", *ptr);
    }
}
```

Em termos de memória



Observe que $\&v[i]$ equivale à ptr !

Observe, ainda, que:



Alocação Dinâmica de Memória I

Exemplo – Alocação 1

```
#include <stdio.h>
main()
{
    int i, s, v[12];
    // Só usa 3 blocos de 4 bytes !
    s = 0;
    for (i=0; i < 3; i++)
    {
        printf(" Entre com v[%d]: ", i);
        scanf("%d", &v[i]);
        // Contando bytes usados.
        s = s + sizeof(v[i]);
    }
    for (i=0; i < 3; i++)
        printf("\n [%4d] \n", v[i]);
    // Mostra o uso da memória.
    printf("Bytes usados = %d \n",s);
    s = sizeof(v);
    printf("Bytes de v = %d \n",s);
}
```

Alocação Estática de Memória p/ v



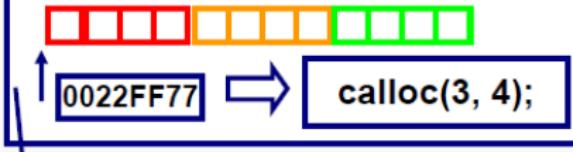
Quando o vetor v é declarado são reservados 12 espaços para elementos do tipo int. Como cada elemento de v é do tipo int e ocupa 4 bytes, então, é realizada a alocação de $12 * 4 = 48$ bytes para o vetor v ! Observe, porém, que só são usados os elementos v[0], v[1] e v[2], ou seja, $3 * 4 = 12$ bytes de um total de 48 bytes.

Alocação Dinâmica de Memória II

Exemplo – Alocação 2

```
#include <stdio.h>
main()
{int i, n; float *nota = NULL;
printf("Entre com tamanho n: ");
scanf("%d",&n);
// Aloca 4*n bytes de memória em
// tempo de execução com
// ponteiro!
nota=(float *)calloc(n,sizeof(float));
for (i=0; i < n; i++)
{
    printf(" Entre a nota %d:", i+1);
    scanf("%f", nota+i);
}
for (i=0; i < n; i++)
    printf("\n [%4f] \n", *(nota+i));
// Mostra o uso da memória.
s = n*sizeof(float);
printf("Bytes de nota = %d \n",s);
}
```

Alocação Dinâmica de Memória



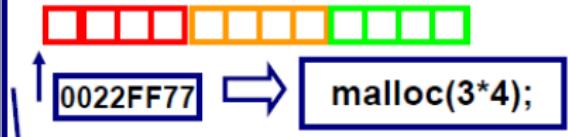
A alocação dinâmica de memória consiste em determinar em tempo de execução o quanto de memória será utilizado. A função `calloc(n, size_t)` fornece o primeiro endereço de memória de um bloco de n espaços com `size_t` bytes. No exemplo acima são alocados $3*\text{sizeof}(\text{float})=3*4 = 12$ bytes.

Alocação Dinâmica de Memória III

Exemplo – Alocação 3

```
#include <stdio.h>
main()
{int i, n; float *nota = NULL;
printf("Entre com tamanho n: ");
scanf("%d",&n);
// Aloca 4*n bytes de memória em
// tempo de execução com
// ponteiro!
nota=(float *)malloc(n*sizeof(float));
for (i=0; i < n; i++)
{
    printf(" Entre a nota %d:", i+1);
    scanf("%f", nota+i);
}
for (i=0; i < n; i++)
    printf("\n [%4f] \n", *(nota+i));
// Mostra o uso da memória.
s = n*sizeof(float);
printf("Bytes de nota = %d \n",s);
}
```

Alocação Dinâmica de Memória



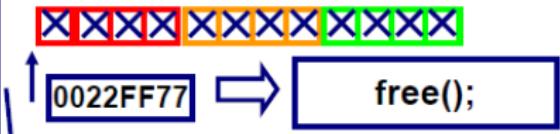
A função `malloc(n*size_t)` fornece o primeiro endereço de memória de um bloco de n espaços com `size_t` bytes. No exemplo acima são alocados $3*sizeof(float) = 3*4 = 12$ bytes. A função `malloc` possui a mesma utilidade da função `calloc`, apenas sua sintaxe é diferente.

Alocação Dinâmica de Memória IV

Exemplo – Alocação 4

```
#include <stdio.h>
main()
{int i, n; float *nota = NULL;
printf("Entre com tamanho n: ");
scanf("%d",&n);
// Aloca 4*n bytes de memória em
// tempo de execução com
// ponteiro!
nota=(float *)malloc(n*sizeof(float));
for (i=0; i < n; i++)
{
    printf(" Entre a nota %d:", i+1);
    scanf("%f", nota+i);
}
for (i=0; i < n; i++)
    printf("\n [%4f] \n", *(nota+i));
// Libera memória alocada cujo
// endereço inicial está em nota.
free(nota);
}
```

Alocação Dinâmica de Memória



Quando é realizada alocação estática de memória o programa em C se encarrega de liberar a memória utilizada.

Mas, quando a alocação dinâmica de memória é empregada, através de malloc ou calloc, é necessário, antes de terminar o uso do programa, liberar a memória através do comando free().

Alocação Dinâmica de Memória V

	Comando	Liberação Memória
Alocação Estática de Memória	<code>int v[3];</code> Reserva 3 espaços de 4 bytes em v	O próprio programa ao ser encerrado, se encarrega de liberar a memória alocada.
Alocação Dinâmica de Memória	<code>v = (int*) malloc (n*sizeof(int));</code> Reserva <i>n</i> espaços de 4 bytes em v	Se não for usar mais a variável <i>v</i> , então, é necessário empregar o comando <code>free()</code> .

Relação entre ponteiros, vetores e matrizes I

- ▶ Assim como é possível alocar memória em tempo de execução para armazenar um vetor, também, é possível construir uma matriz M com m linhas e n colunas. Os comandos para tal tarefa são como dados a seguir:

```
int main()
{
    char **M,
    int m, n, i;
    printf("Entre com m e n");
    scanf("%d %d", &m, &n);
    // Vetor de endereços (os elementos são do tipo char*)
    M = (char**) calloc (m, sizeof(char *));
    // Cria para cada endereço um vetor de elementos int
    for (i = 0; i < m; i++)
        M[i] = (char*) calloc (n, sizeof(char))
    return 0;
}
```

Relação entre ponteiros, vetores e matrizes II

- ▶ Vejamos um exemplo:

```
M = (char**) calloc (3, sizeof(char*));
```

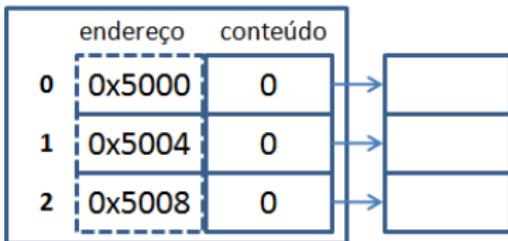
M → NULL

Relação entre ponteiros, vetores e matrizes III

```
M = (char**)calloc(3, sizeof(char))
```

M

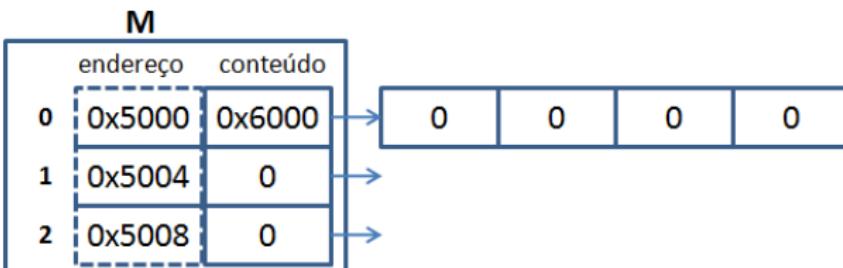
	endereço	conteúdo
0	0x5000	0
1	0x5004	0
2	0x5008	0



The diagram illustrates the state of memory after the allocation. A pointer variable M is shown at the top, pointing to the first element of a dynamically allocated array. The array itself is represented by a table with three rows, indexed 0, 1, and 2. Each row contains two columns: 'endereço' (address) and 'conteúdo' (content). The addresses are 0x5000, 0x5004, and 0x5008 respectively. All content cells contain the value 0. Three blue arrows point from the right side of the table to the right side of the next column, indicating the start of three separate memory blocks.

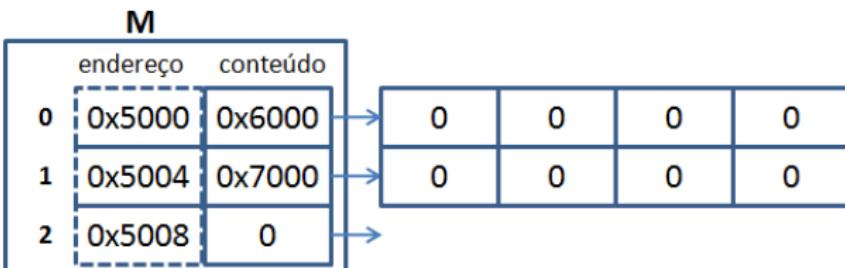
Relação entre ponteiros, vetores e matrizes IV

`M[0] = (char*)calloc(4, sizeof(char))`

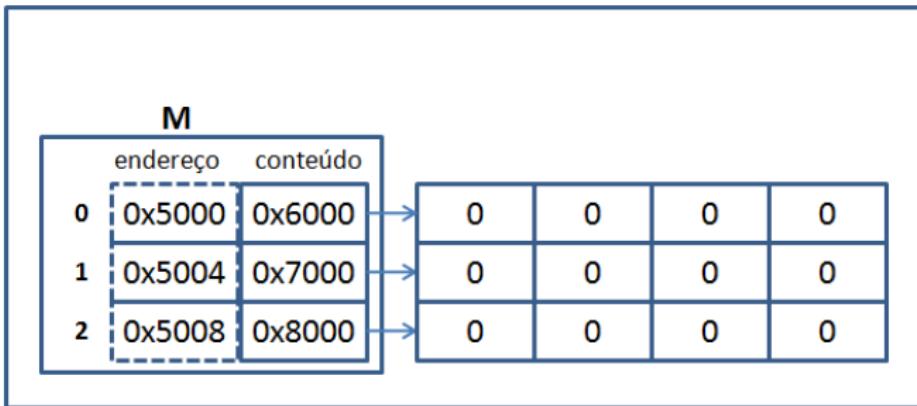


Relação entre ponteiros, vetores e matrizes V

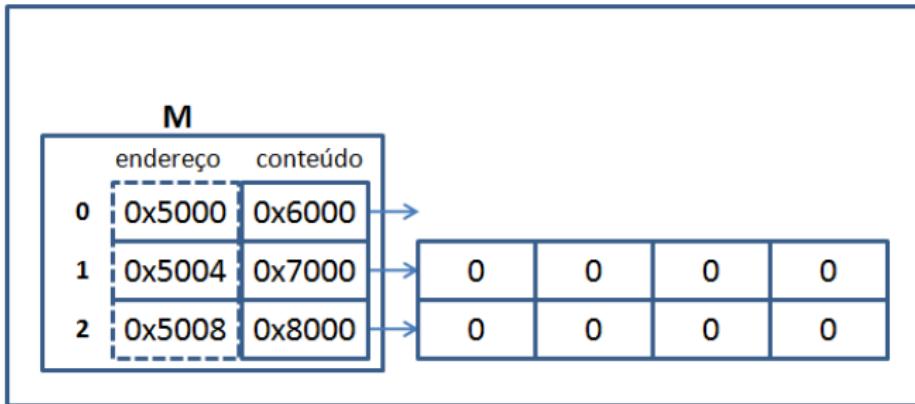
`M[2] = (char*)calloc(4, sizeof(char))`



Relação entre ponteiros, vetores e matrizes VI

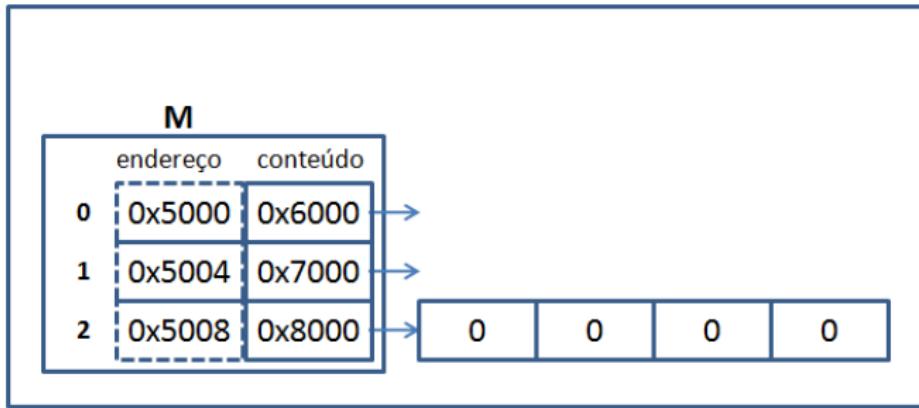


Liberação de memória I

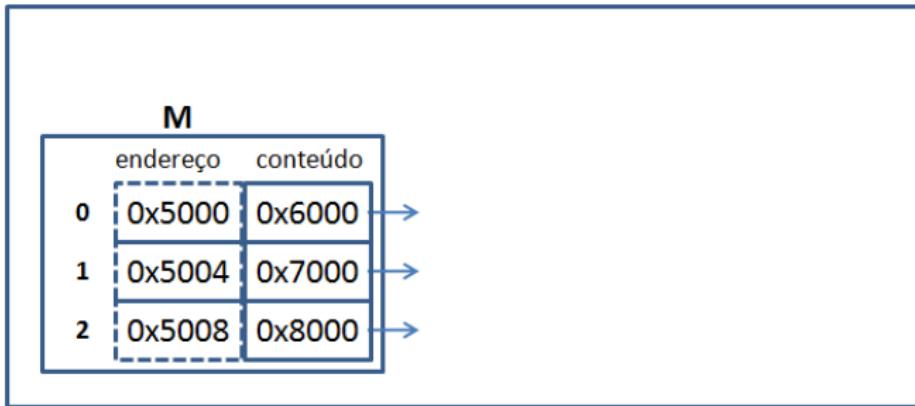


`free(M[0])`

Liberação de memória II



Liberação de memória III



$free(M[2])$

Liberação de memória IV

M

free(M)

Exercícios I

Inserir n notas de um total de m alunos

Exercícios II

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i, j, m, n; float **M = NULL;
    printf("Entre com m e n: ");
    scanf("%d %d", &m, &n);
    // Aloca m espaços tipo float *.
    M = (float **)calloc(m, sizeof(float *));
    // Aloca n espaços tipo float, cada M[i].
    for (i = 0; i < m; i++)
        M[i] = (float *)calloc(n, sizeof(float));
    // Preenchendo a matriz M usando índices: M[i][j].
    for (i=0; i < m; i++)
    {
        printf("Aluno %d: ", i+1);
        for (j=0; j < n; j++)
        {
            printf(" Nota %d:", j+1);
            scanf("%f", &M[i][j]);
        }
    }
    . . .
}
```

Exercícios III

```
// Impressão dos elementos de M, empregando ponteiros.  
for ( i=0; i < m; i++)  
{    for ( j=0; j < n; j++)  
        printf(" [%4f] ", M[ i ][ j ]);  
        printf(" \n ");  
}  
// Liberação de memória.  
// Liberando m vetores de tamanho n.  
if (M != NULL){  
    for ( i=0; i < m; i++)  
        if (M[ i ] != NULL) free (M[ i ]);  
    // Liberando o vetor de ponteiros  
    // de tamanho m.  
    free (M);  
}  
return 0;  
}
```

Ponteiros: Passagem por valor e por referência I

Implementar uma função que encontre o maior de dois números

Exemplo 10: Programa Maior

```
#include <stdlib.h>
#include <stdio.h>
int Maior(int x, int y)
{
    return (x > y) ? x:y;
}
main()
{
    int a, b, max;
    printf("Entre com a e b:");
    scanf("%d %d", &a, &b);
    max = Maior(a,b);
    printf("Max{a,b} = %d \n",max);
}
```

Escopo da Função Main

a	b
4	5
5000	5050

Escopo da Função Maior

x	y
4	5
6000	7050
return 5;	

Ponteiros: Passagem por valor e por referência II

Exemplo 11: Programa Troca

```
#include <stdlib.h>
#include <stdio.h>
int Troca( int *x, int *y )
{
    int tmp = *x;
    *x = *y;
    y = tmp;
}
main()
{int a, b, max;
printf("Entre com a e b:");
scanf("%d %d", &a, &b);
printf("a=%d - b=%d\n",a,b);
Troc(a, &b);
printf("a=%d - b=%d\n",a,b);}

```

Escopo da Função main

a	b
4	5
5000	5050

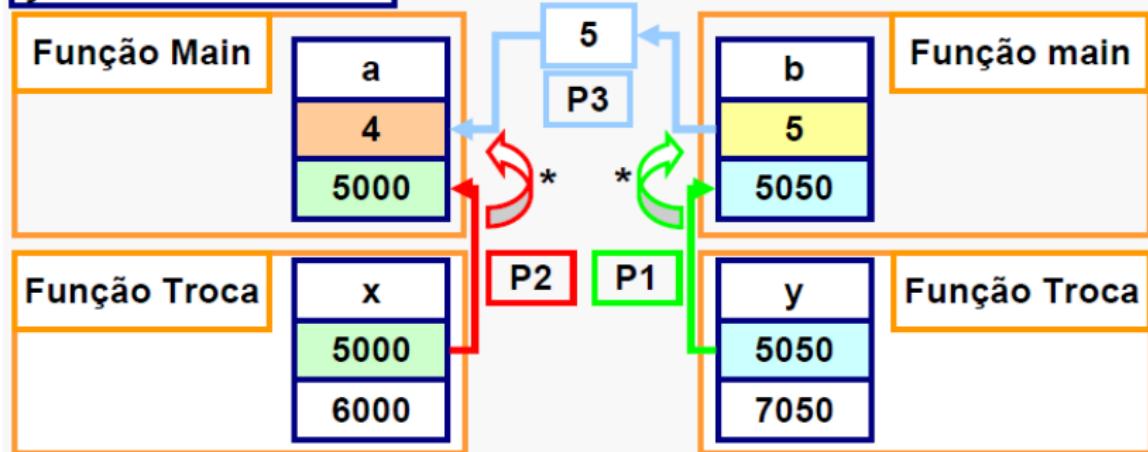
Escopo da Função Troca

x	y
5000	5050
6000	7050

Ponteiros: Passagem por valor e por referência III

```
int Troca(int *x, int *y)
{
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

O comando é realizado em três passos:
(P1) Com $*y$ obtém-se o valor contido em b.
(P2) Com $*x$ obtém-se o valor contido em a.
(P3) O valor de b (5) é passado para a.



Ponteiros: Passagem por valor e por referência IV

Exemplo 12–Referência ou Valor

```
#include <stdio.h>
void printV(int X[], int n)
{ int i;
  for (i = 0; i < 10; i++)
    printf(" [%-4d] ",X[i]); puts("");
}

void preencheV(int X[], int n)
{ int i;
  for (i = 0; i < 10; i++)
    X[i] = 2*i + 1;
}

main ( )
{ int i, n=10, V[10] = {0};
  printV(V, n);
  preencheV(V, n);
  printV(V, n);
}
```

Escopo da Função main

V[0]	V[1]	...	V[9]
0	0	...	0
5000	5004	...	5036

Função preencheV

Escopo da Função preencheV

X[0]	X[1]	...	X[9]
1	3	...	17
5000	5004	...	5036

Ponteiros: Passagem por valor e por referência V

Criar uma estrutura empregado com os seguintes campos:

- ▶ nome
- ▶ salario
- ▶ sexo

Inserir n empregados (criar um vetor dinâmico)

Ponteiros: Passagem por valor e por referência VI

```
typedef struct Pessoa
{
    char nome[100];
    double salario;
    char sexo;
}PE;

void Insere(PE*, int);
void Print(PE*, int);
int main()
{
    int n;
    PE *trab = NULL;
    printf("Quantidade de pessoas");
    scanf("%d%*c", &n);
    trab = (PE*) calloc(n, sizeof(PE));
    Insere(trab, n);
    Print(trab, n);
    return 0;
}
```

Ponteiros: Passagem por valor e por referência VII

```
void Insere(PE* vet, int n)
{
    int i, valores;
    for (i = 0; i < n; i++)
    {
        printf("Cadastro numero %d\n", i+1);
        printf("Insere nome: ");
        fgets(vet[i].nome, 100, stdin);
        do{
            printf("Insere salario: ");
            valores = scanf("%lf", &vet[i].salario);
            if (valores == 0) scanf("%*s");
        }while(valores == 0);

        printf("Insere sexo: ");
        scanf("%c%*c", &vet[i].sexo);
        //vet[i].sexo = getchar();
    }
}
```

Ponteiros: Passagem por valor e por referência VIII

```
void Print(PE* vet, int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        printf("%s %lf %c \n",
               vet[i].nome, vet[i].salario, vet[i].sexo);
    }
}
```

Exercícios em Propostos I

1. Defina uma função que retorne a transposta de uma matriz x de dimensão $lin \times col$.

```
int** Transposta(int** M, int lin, int col)
```

2. Elaborar um programa para calcular a média aritmética de dois valores reais utilizando apenas variáveis do tipo ponteiro.

FIM