BCC 201 - Introdução à Programação I

Ponteiros

Guillermo Cámara-Chávez
UFOP

Memoria I

Endereço	Valor
00000000	??
00000001	??
00000002	??
00000003	??
00000004	??
00000005	??
00000006	??
00000007	??
00000008	??
00000009	??
0000000A	??
0000000B	??
0000000C	??
0000000D	??

- A memória está formada por várias células.
- Cada célula contém um endereço e um valor.
- O tamanho do endereço e o tamanho do valor dependem da arquitetura do computador (32/64 bits)

Endereço	Valor
0000000D	??

Memoria II

Endereço	Valor
00000000	??
00000001	??
00000002	??
00000003	??
00000004	??
00000005	??
00000006	??
00000007	??
00000008	??
00000009	??
0000000A	??
0000000B	??
0000000C	??
0000000D	??

} i

```
int main()
{
 → char i;
    return 0;
}
```

- Declaro um caracter chamado *i*.
- Os caracteres ocupam 1 byte na memória (para uma arquitetura de 32 bits)

Memoria III

Endereço	Valor
00000000	??
00000001	
00000002	
00000003	
00000004	??
00000005	??
00000006	??
00000007	??
00000008	??
00000009	??
0000000A	??
0000000B	??
0000000C	??
0000000D	??

i

```
int main()
{
 → int i;
    return 0;
}
```

- Declaro um número inteiro chamado *i*.
- Os inteiros ocupam 4 bytes na memória (para uma arquitetura de 32 bits)

Memoria IV

Endereço	Valor
00000000	??
00000001	
00000002	
00000003	
00000004	??
00000005	??
00000006	??
00000007	??
00000008	??
00000009	??
0000000A	??
0000000B	??
0000000C	??
0000000D	??

i

```
int main()
{
 → float i;
    return 0;
}
```

- Declaro um número ponto flutuante chamado *i*.
- Os flutuantes ocupam 4 bytes na memória (para uma arquitetura de 32 bits)

Memoria V

Endereço	Valor
00000000	??
00000001	
00000002	
00000003	
00000004	
00000005	
00000006	
00000007	
00000008	??
00000009	??
0000000A	??
0000000B	??
0000000C	??
0000000D	??

i

```
int main()
{
  → double i;
    return 0;
}
```

- Declaro um número flutuante de dupla precisão chamado *i*.
- Os flutuantes de dupla precisão ocupam 8 bytes na memória (para uma arquitetura de 32 bits)

Memoria VI

Endereço	Valor	
00000000	??	c
00000001		
00000002		
00000003		
00000004	??	i
00000005		
00000006		
00000007		
00000008	??	f
00000009		
0000000A		
0000000B		
0000000C	??	d
0000000D		
0000000E		
0000000F		

```
int main()
{
 → char* c;
 → int* i;
 → float* f;
 → double* d;
    return 0;
}
```

- Declaração de quatro ponteiros(*c*, *i*, *f* e *d*). Cada ponteiro de um tipo diferente(*char*, *int*, *float*, *double*).
- Todos eles ocupam o mesmo espaço na memória, 4 bytes.
- Isso acontece porque todos eles armazenam endereços de memória, e o tamanho de um endereço de memória é o mesmo para todos os tipos.

Memoria VII

Endereço	Valor
00000000	??
00000001	
00000002	
00000003	
00000004	??
00000005	??
00000006	??
00000007	??
00000008	??
00000009	??
0000000A	??
0000000B	??
0000000C	??
0000000D	??
0000000C	??
0000000D	??

i

```
int main()
{
 → int i;
    i = 15;
    char c = 's';
    int *p = &i;
    *p = 25;
    return 0;
}
```

- Declaração de um inteiro *i*.

Memoria VIII

Endereço	Valor
00000000	15
00000001	
00000002	
00000003	
00000004	??
00000005	??
00000006	??
00000007	??
00000008	??
00000009	??
0000000A	??
0000000B	??
0000000C	??
0000000D	??
0000000C	??
0000000D	??

i

```
int main()
{
    int i;
 → i = 15;
    char c = 's';
    int *p = &i;
    *p = 25;
    return 0;
}
```

- A variável *i* recebe o valor 15. Esse valor 15 é colocado no campo valor da memória alocada previamente para a variável i.
- Lembrem que essa notação com o 15 na ultima casa é apenas didática na verdade esse valor é tudo em binário.

Memoria IX

Endereço	Valor	
00000000	15	i
00000001		
00000002		
00000003		
00000004	s	} c
00000005	??	
00000006	??	
00000007	??	
00000008	??	
00000009	??	
0000000A	??	
0000000B	??	
0000000C	??	
0000000D	??	
0000000C	??	
0000000D	??	

```
int main()
{
    int i;
    i = 15;
→   char c = 's';
    int *p = &i;
    *p = 25;
    return 0;
}
```

- A variável *c* do tipo `char` é criada e inicializada com o valor 's'.

Memoria X

Endereço	Valor	
→00000000	15	i
00000001		
00000002		
00000003		
00000004	s	} c
00000005	00	} p
00000006	00	} p
00000007	00	} p
00000008	00	} p
00000009	??	
0000000A	??	
0000000B	??	
0000000C	??	
0000000D	??	
0000000C	??	
0000000D	??	

```
int main()
{
    int i;
    i = 15;
    char c = 's';
 → int *p = &i;
    *p = 25;
    return 0;
}
```

- Ponteiro de inteiro declarado.
- O nome desse ponteiro é *p* e ele é inicializada no momento de sua criação.
- O valor que esse ponteiro recebe é o endereço da variável *i*(&*i*) que nesse caso é o endereço 00000000.
- Dizemos que *p* aponta para *i*.

Memoria XI

Endereço	Valor	
00000000	25	i
00000001		
00000002		
00000003		
00000004	s	} c
00000005	00	} p
00000006	00	} p
00000007	00	} p
00000008	00	} p
00000009	??	
0000000A	??	
0000000B	??	
0000000C	??	
0000000D	??	
0000000C	??	
0000000D	??	

```
int main()
{
    int i;
    i = 15;
    char c = 's';
    int *p = &i;
 → *p = 25;
    return 0;
}
```

- Finalizando, fazemos uma atribuição.
- Colocamos 25 no valor apontado por *p*. Como visto no slide anterior *p* aponta para *i*
- Desse modo, colocamos 25 no valor da variável *i*.

Endereços I

```
int x = 100;
```

1. Ao declararmos uma variável x como acima, temos associados a ela os seguintes elementos:
   - Um nome (x)
   - Um endereço de memória ou referência (0xbfd267c4)
   - Um valor (100)
2. Para acessarmos o endereço de uma variável, utilizamos o operador &

Endereços II

3. Um ponteiro (apontador ou *pointer*) é um tipo especial de variável cujo valor é um endereço
4. Um ponteiro pode ter o valor especial nullptr, quando não contém nenhum endereço.
5. nullptr é usado para inicializar um ponteiro

Endereços III

```
*var
```

6. A expressão acima representa o conteúdo do endereço de memória guardado na variável var.
7. Ou seja, var não guarda um valor, mas sim um endereço de memória.

O operador address-of (&) I

- Retorna o endereço de uma variável

```
int main()
{
    int x = 100;
    cout << "O valor de x = " << x << endl;
    cout << "O endereco de x = " << &x << endl;
    return 0;
}
```

Aparece na tela:
O valor de $x = 100$
O endereco de $x =$ 0xbfd267c4

Endereço	Valor	Nome
bfd267c4	100	**x**

Ponteiro I

- Um apontador é uma variável que pode armazenar endereços de outras variáveis

```
int x;
int *px; //apontador para inteiros
px = &x; //px aponta a x
```

Ponteiro II

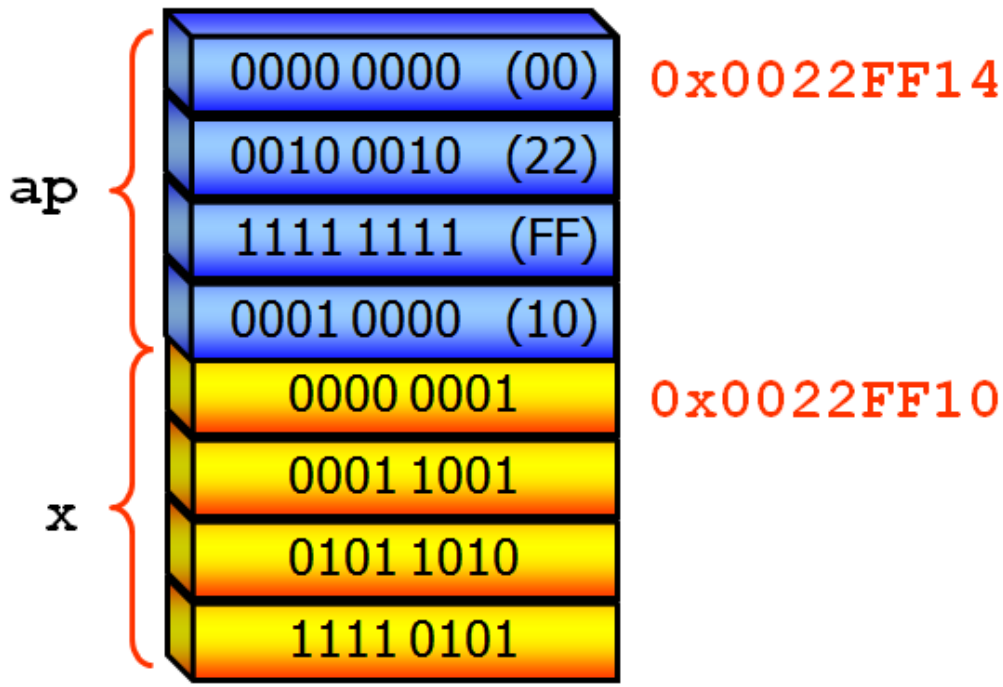
```
int main() {
  int x = 100;
  int *p_x = &x;

  cout << "valor de x = "    << x   << endl;
  cout << "endereço de x = " << &x  << endl;
  cout << "endereço de x = " << p_x << endl;

  return 0;
}
```

Aparece na tela:
valor de $x = 100$
endereço de x = 0x0022ff38
endereço de x = 0x0022ff38

Endereço	Valor	Nome
0022ff38	100	**x**
0022ff3c	0022ff38	p_x

Ponteiro III

```
int x;
int *ap;     // apontador para inteiros
ap = &x;     // ap aponta para x
```



Declaração de ponteiros I

- Há vários tipos de ponteiros:
  - ponteiros para caracteres
  - ponteiros para inteiros
  - ponteiros para ponteiros para inteiros
  - ponteiros para vetores
  - ponteiros para estruturas
- O compilador C++ faz questão de saber de que tipo de ponteiro você está definindo.

Declaração de ponteiros II

- Utilizamos o operador unário *

```
int *p_int;
char *p_char;
float *p_float;
double *p_double;
```

Declaração de ponteiros III

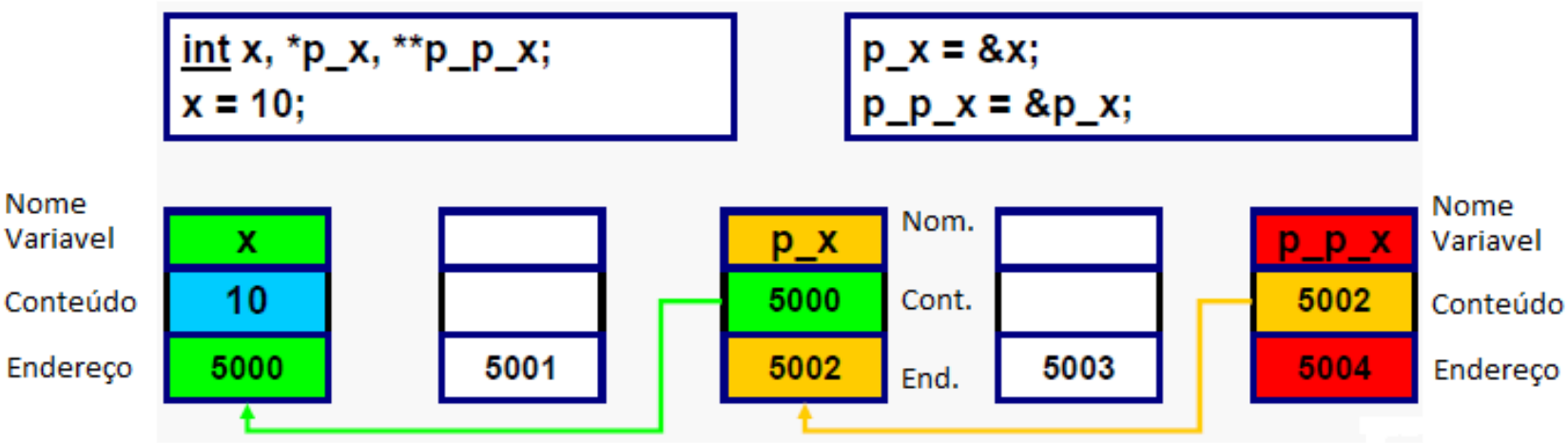
- Para declarar vários apontadores em uma única linha:

```
int *ap1, *ap2, *ap3;
```

Declaração de ponteiros IV

- Definição - Ponteiros para Ponteiros (tipo **ponteiro)
- As variáveis do tipo ponteiro ocupam um endereço de memória
- É possível criar uma nova variável para também armazenar este endereço
- Ou seja, é possível criar ponteiros para ponteiros.

Declaração de ponteiros V



Declaração de ponteiros VI

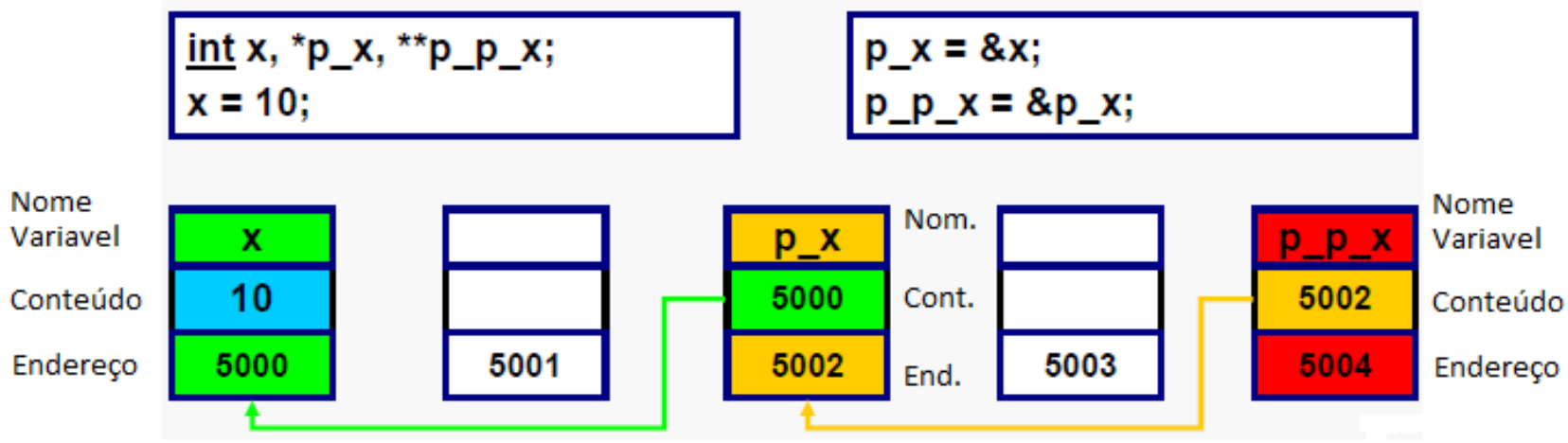
```
int main() {
  int **pp_int, *p_int, i = 100;

  p_int = &i;
  pp_int = &p_int;

  cout << " Nome | Endereco | Valor \n");
  cout << "     i| " << &i      << " | " << i      << endl;
  cout << " p_int| " << &p_int  << " | " << p_int  << endl;
  cout << "pp_int| " << &pp_int << " | " << pp_int << endl;

  return 0;
}
```

Aparece na tela:

Nome	Endereco	Valor
i	0x22ff3c	100
p_int	0x22ff40	0x22ff3c
pp_int	0x22ff44	0x22ff40

Declaração de ponteiros VII

- Exemplo de utilização de ponteiros e os operadores de endereço (&) e de referência ($*$)



Fazendo acesso aos valores das variáveis referenciadas I

```
int x;
int* px = &x;

*px = 3;
```

*px pode ser usado em qualquer contexto que x seria.

Fazendo acesso aos valores das variáveis referenciadas II

Endereço	Conteúdo	Nome
...		
0x1000		
0x1001		
0x1002		
0x1003		
0x1004		
0x1005		
0x1006		
0x1007		
0x1008		
0x1009		
...		

```
int x;
int* px;
px = &x;
*px = 3;
```

Fazendo acesso aos valores das variáveis referenciadas III

Endereço	Conteúdo	Nome
...		
0x1000		
0x1001		
0x1002		x
0x1003		
0x1004		
0x1005		
0x1006		
0x1007		
0x1008		
0x1009		
...		

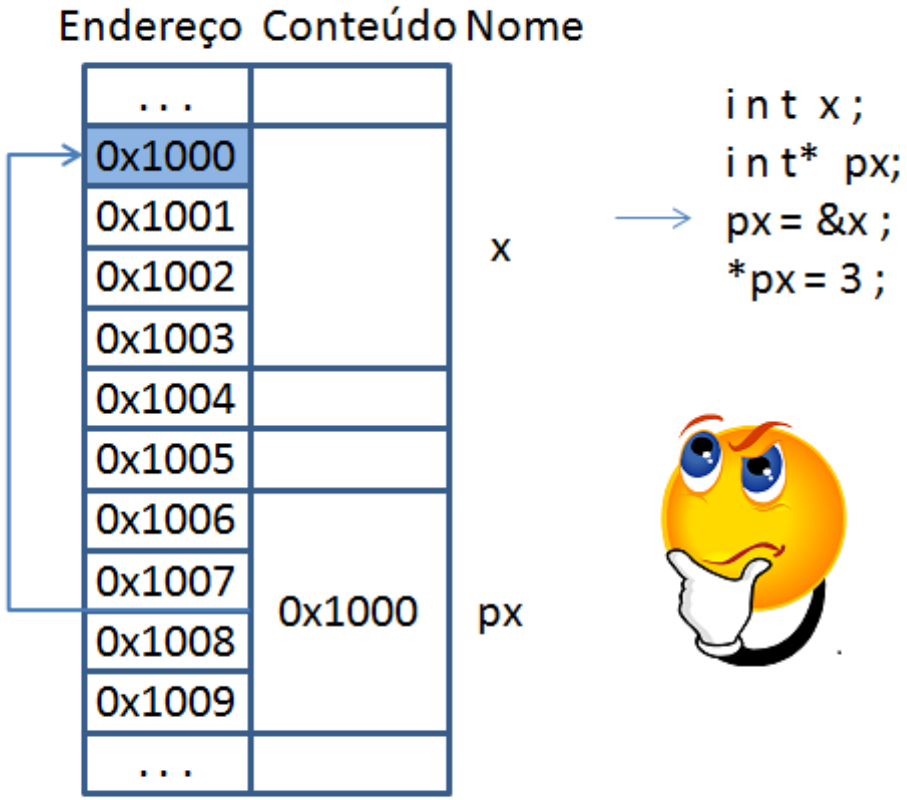
```
int x;
int* px;
px = &x;
*px = 3;
```

Fazendo acesso aos valores das variáveis referenciadas IV

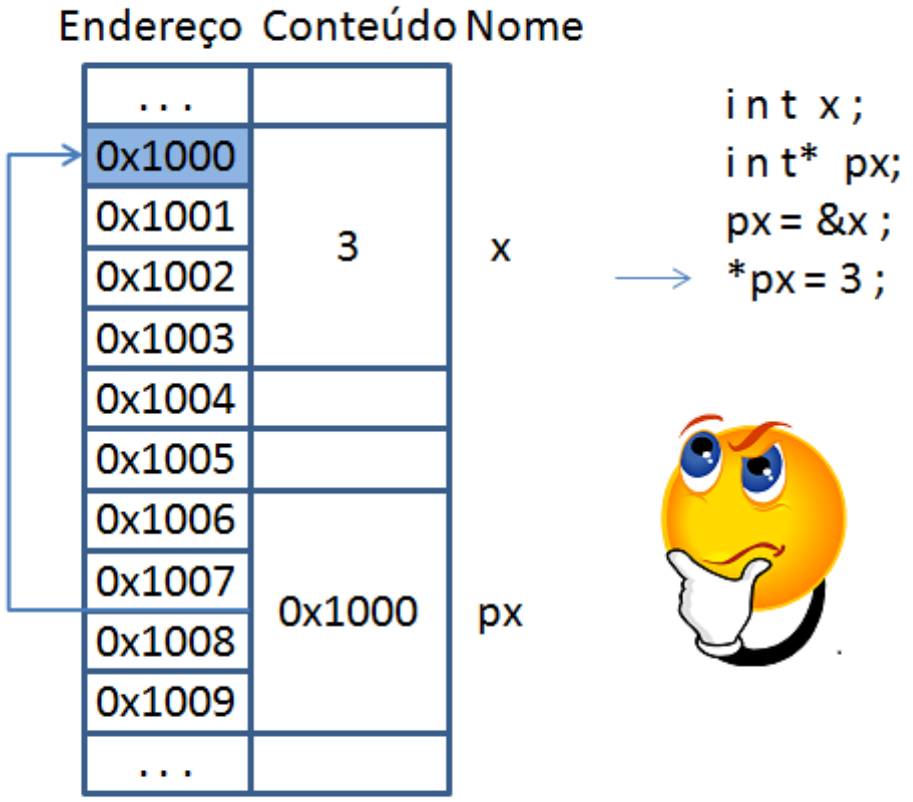
Endereço	Conteúdo	Nome
...		
0x1000		
0x1001		x
0x1002		
0x1003		
0x1004		
0x1005		
0x1006		
0x1007		px
0x1008		
0x1009		
...		

```
int x;
int* px;
px = &x;
*px = 3;
```

Fazendo acesso aos valores das variáveis referenciadas V



Fazendo acesso aos valores das variáveis referenciadas VI



Fazendo acesso aos valores das variáveis referenciadas VII

```
int main() {
  int x;
  int *p_x = &x;

  x = 100;
  cout << "valor de x = " << *p_x << endl;

  *p_x = 200;
  printf("valor de x = " << *p_x << endl;

  return 0;
}
```

Aparece na tela:
valor de x = 100
valor de x = 200

Passagem de parâmetros por valor I

```
void nao_troca(int x, int y)
{
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}
```

Apenas uma cópia de x e y é passada para a função

Passagem de apontadores I

```
void troca(int *x, int *y)
{
    int tmp;

    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

Passagem de apontadores II

Comparando ambas funções . . .

```
void nao_troca(int x, int y) {
  int aux;

  aux = x;
  x = y;
  y = aux;
}

void troca(int *ap_x, int *ap_y) {
  int aux;

  aux = *ap_x;
  *ap_x = *ap_y;
  *ap_y = aux;
}
```

Passagem de apontadores III

```
int main() {
  int x = 100, y = 200;

  nao_troca(x, y);
  cout << "x = " << x << " y = " << y << endl;

  troca(&x, &y);
  cout << "x = " << x << " y = " << y << endl;

  return 0;
}
```

Mostra na tela:

x = 100 y = 200

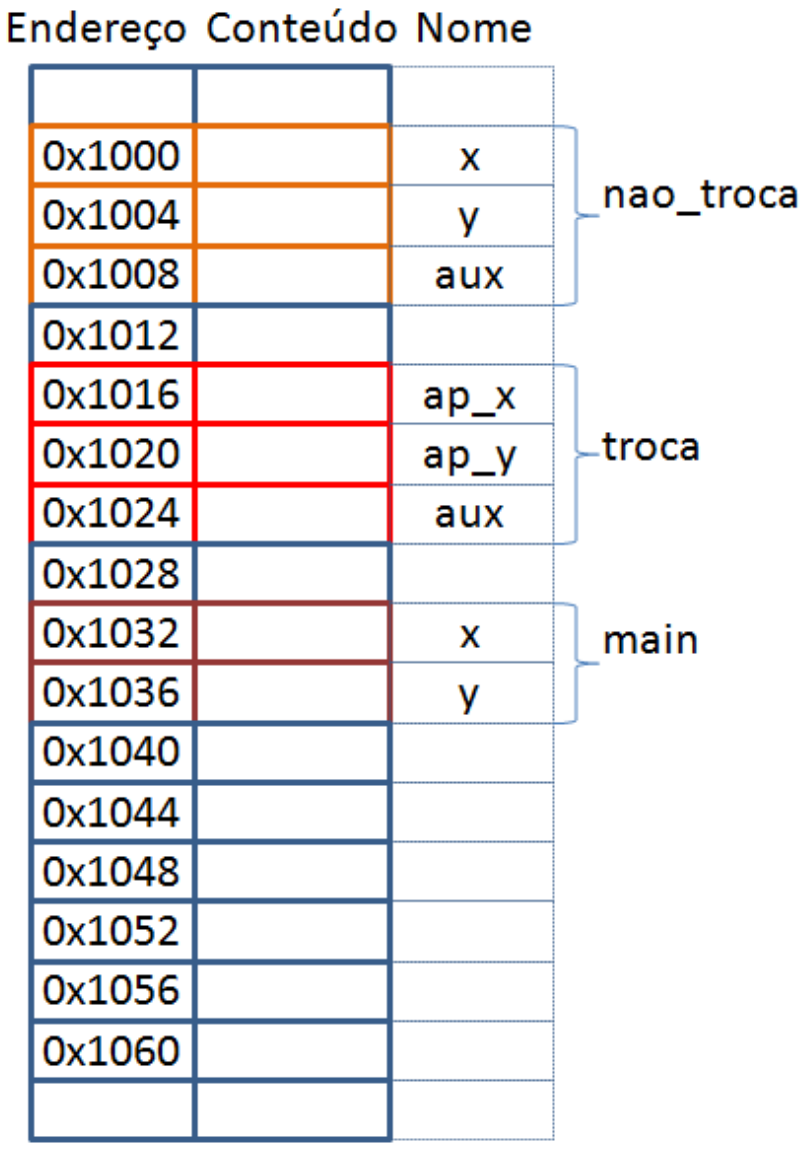
x = 200 y = 100

Passagem de apontadores IV

Endereço	Conteúdo	Nome
0x1000		
0x1004		
0x1008		
0x1012		
0x1016		
0x1020		
0x1024		
0x1028		
0x1032		
0x1036		
0x1040		
0x1044		
0x1048		
0x1052		
0x1056		
0x1060		

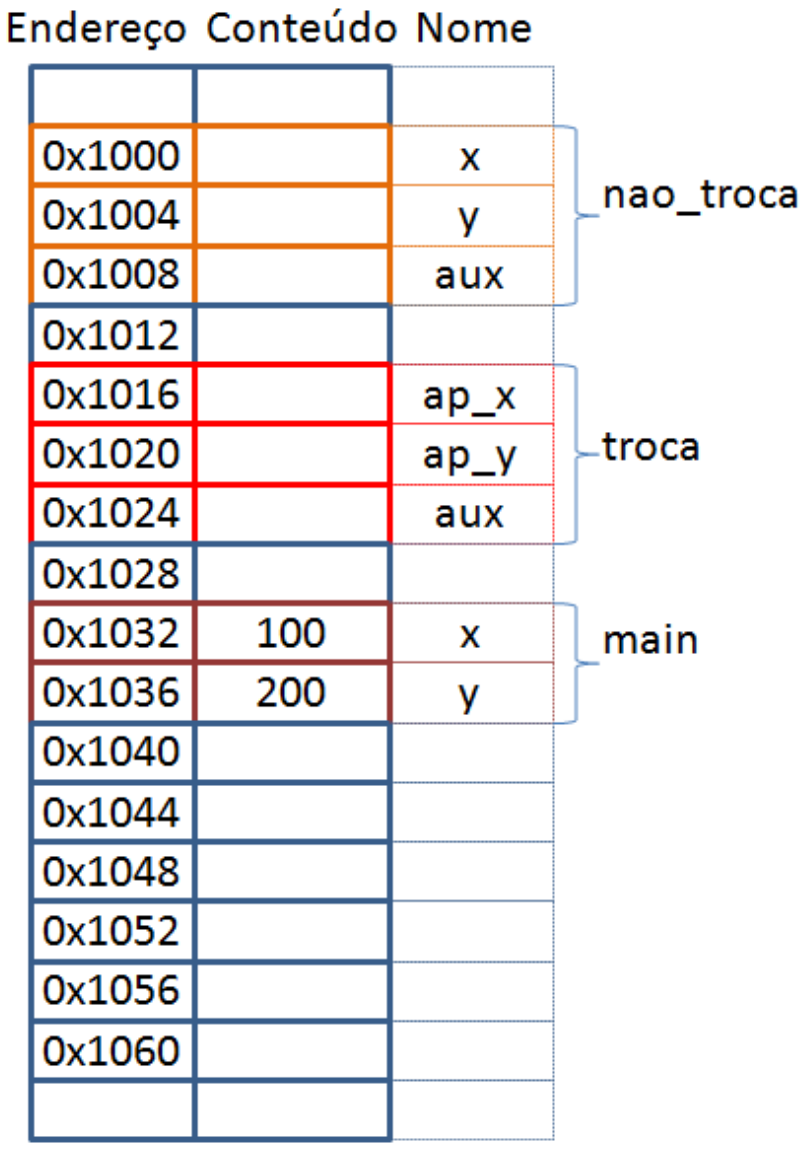
```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores V




```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_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;
}
```

Passagem de apontadores VI



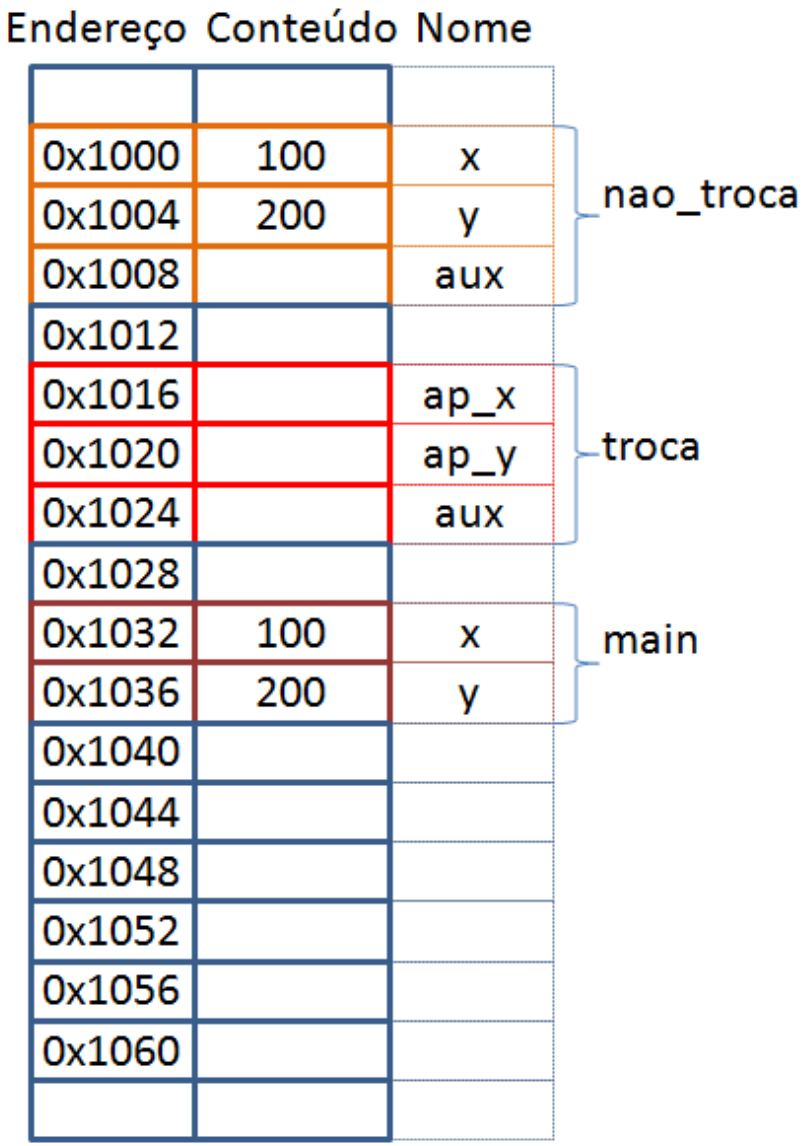

```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores VII

Endereço	Conteúdo	Nome	
0x1000		x	nao_troca
0x1004		y	
0x1008		aux	
0x1012			
0x1016		ap_x	troca
0x1020		ap_y	
0x1024		aux	
0x1028			
0x1032	100	x	main
0x1036	200	y	
0x1040			
0x1044			
0x1048			
0x1052			
0x1056			
0x1060			

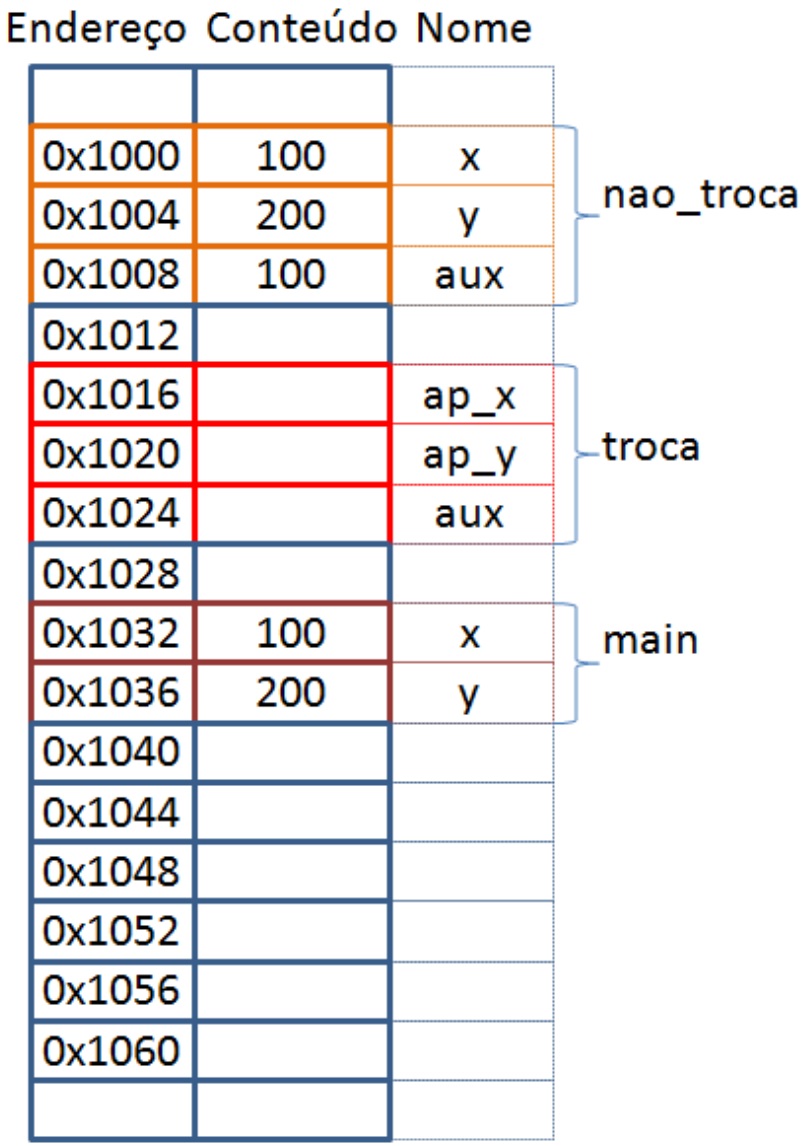
```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores VIII



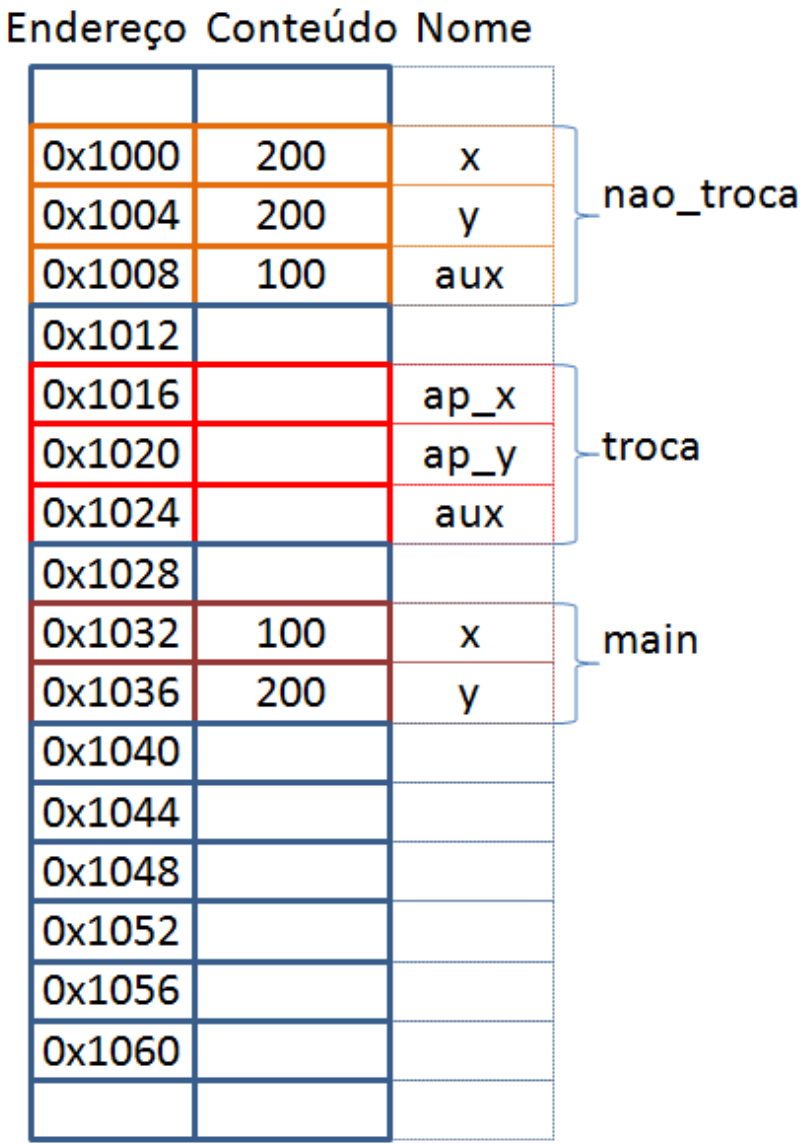

```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores IX




```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_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;
}
```

Passagem de apontadores X




```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_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;
}
```

Passagem de apontadores XI

Endereço	Conteúdo	Nome	
0x1000	200	x	nao_troca
0x1004	100	y	
0x1008	100	aux	
0x1012			
0x1016		ap_x	troca
0x1020		ap_y	
0x1024		aux	
0x1028			
0x1032	100	x	main
0x1036	200	y	
0x1040			
0x1044			
0x1048			
0x1052			
0x1056			
0x1060			

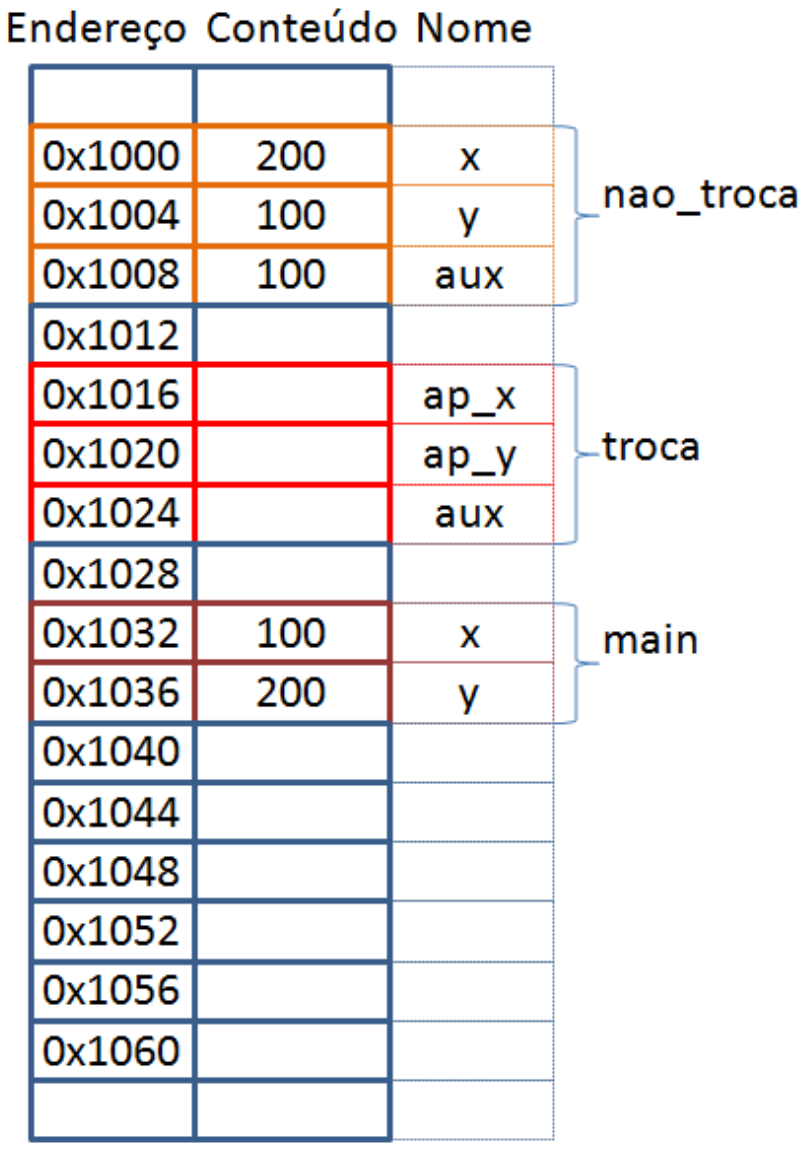
```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores XII

Endereço	Conteúdo	Nome	
0x1000	200	x	nao_troca
0x1004	100	y	
0x1008	100	aux	
0x1012			
0x1016		ap_x	troca
0x1020		ap_y	
0x1024		aux	
0x1028			
0x1032	100	x	main
0x1036	200	y	
0x1040			
0x1044			
0x1048			
0x1052			
0x1056			
0x1060			

```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_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;
}
```

Passagem de apontadores XIII



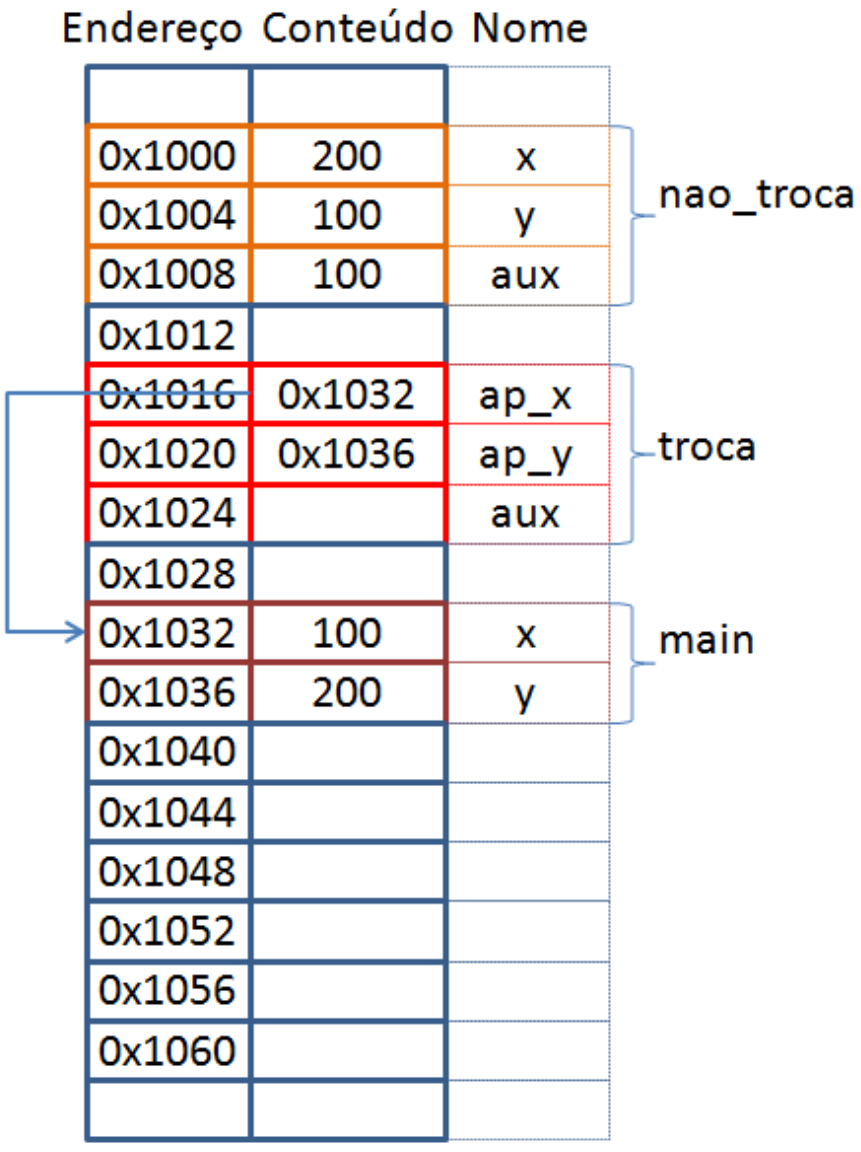

```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores XIV

Endereço	Conteúdo	Nome	
0x1000	200	x	nao_troca
0x1004	100	y	
0x1008	100	aux	
0x1012			
0x1016	0x1032	ap_x	troca
0x1020	0x1036	ap_y	
0x1024		aux	
0x1028			
0x1032	100	x	main
0x1036	200	y	
0x1040			
0x1044			
0x1048			
0x1052			
0x1056			
0x1060			

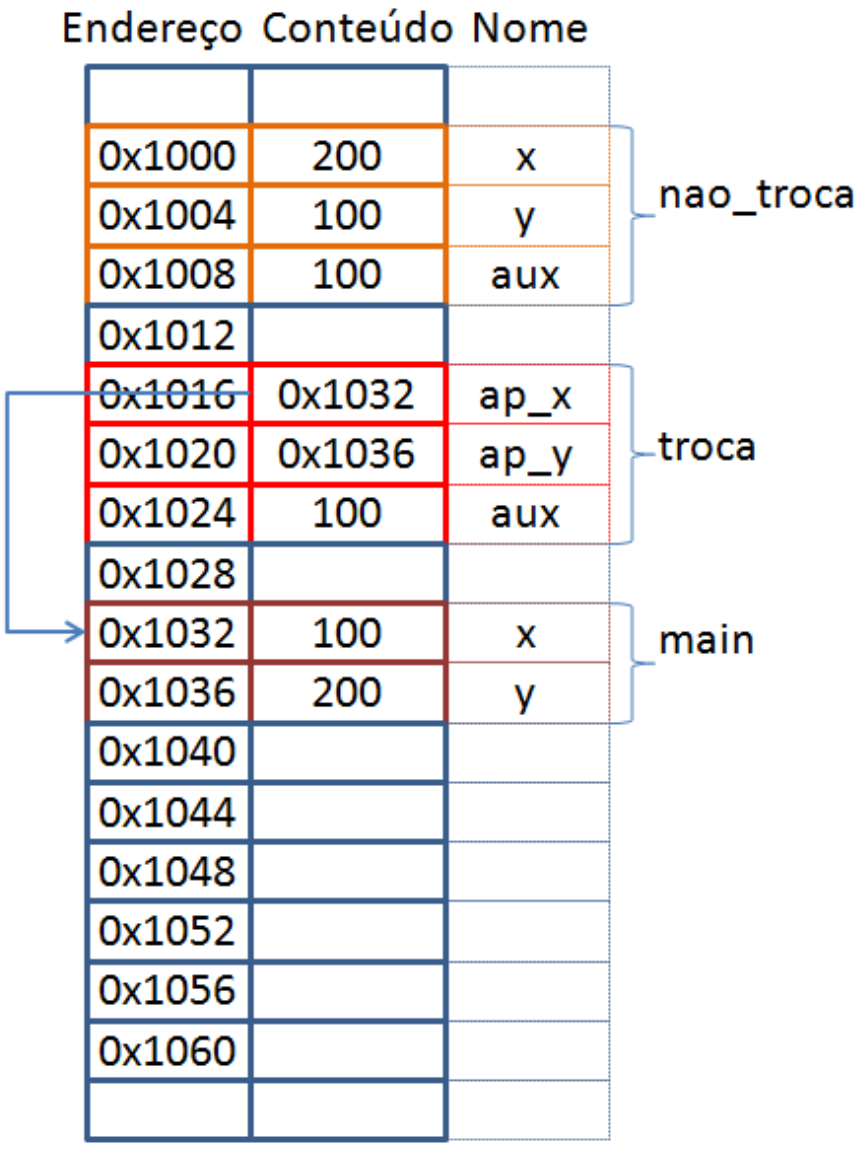
```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores XV



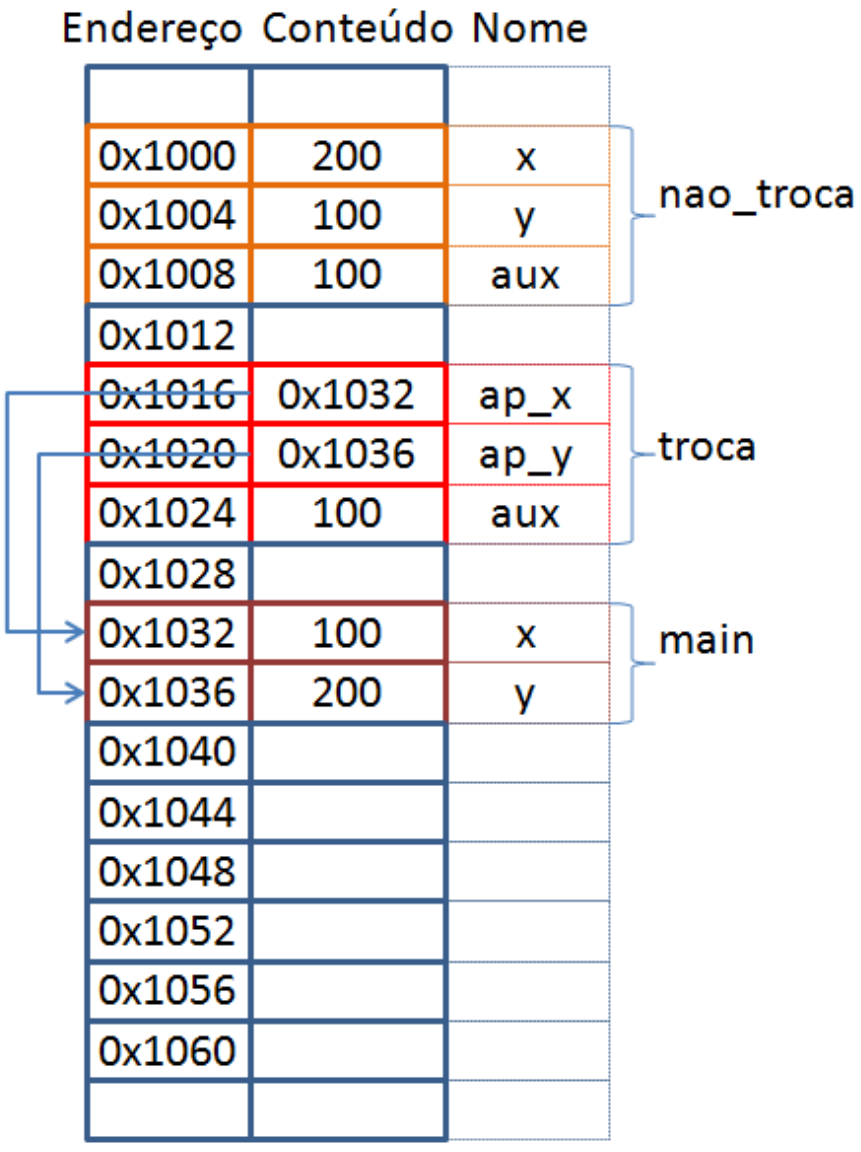

```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores XVI



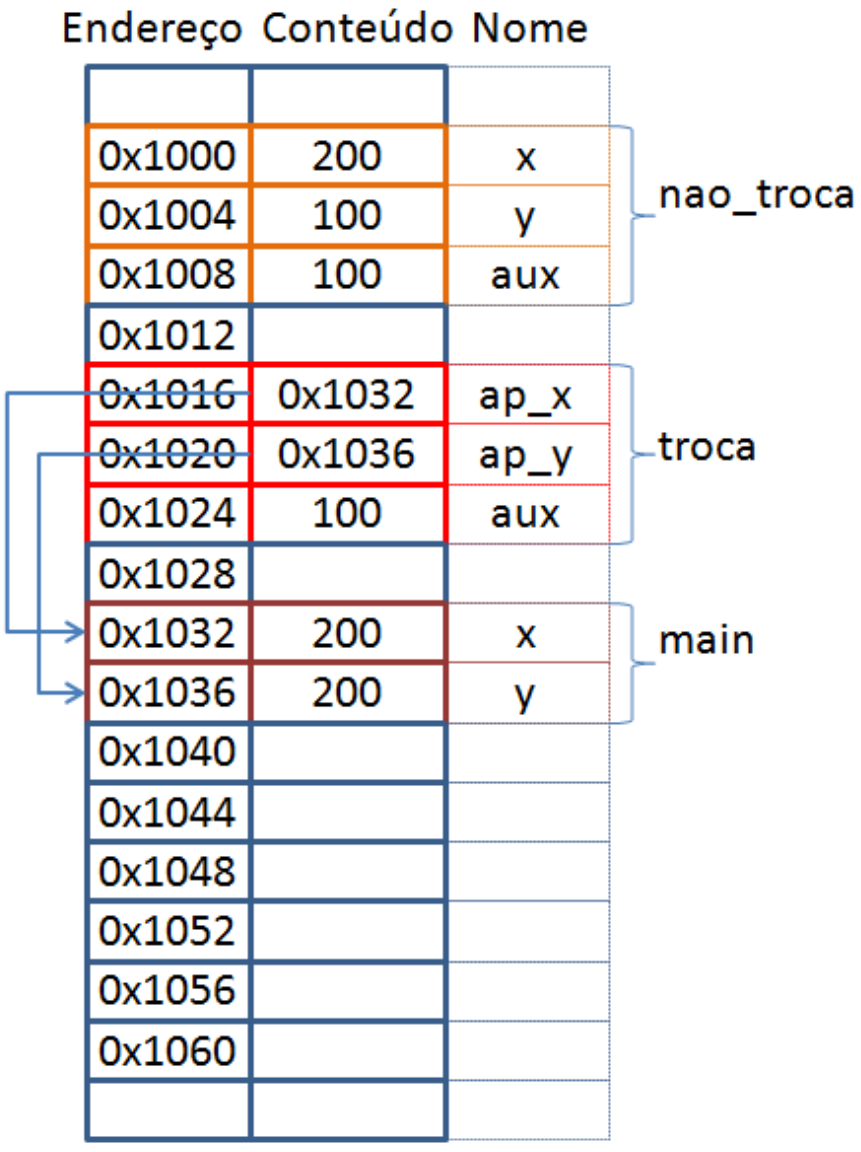

```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores XVII



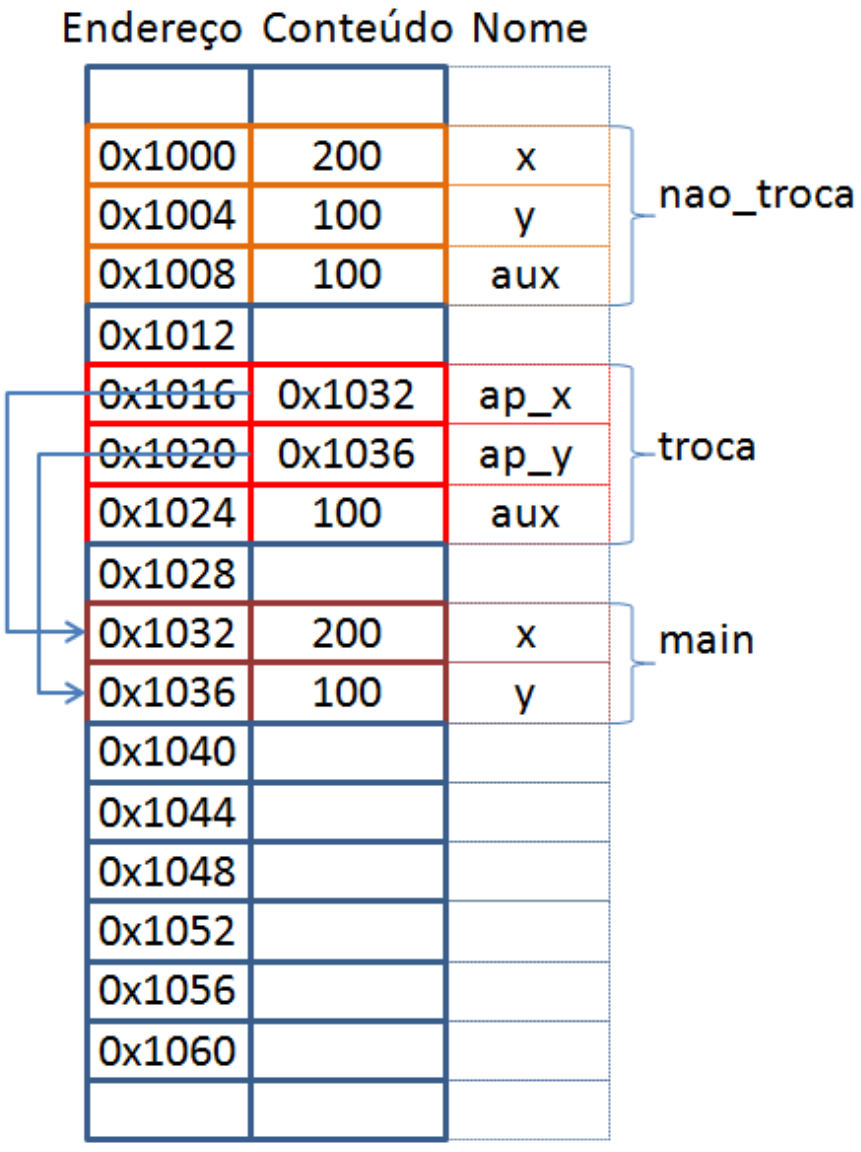

```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 printf("x = %d y = %d \n", x, y);
 troca(&x, &y);
 printf("x = %d y = %d \n", x,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

Passagem de apontadores XVIII




```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_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;
}
```

Passagem de apontadores XIX




```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_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;
}
```

Passagem de apontadores XX

Endereço	Conteúdo	Nome	
0x1000	200	x	nao_troca
0x1004	100	y	
0x1008	100	aux	
0x1012			
0x1016	0x1032	ap_x	troca
0x1020	0x1036	ap_y	
0x1024	100	aux	
0x1028			
0x1032	200	x	main
0x1036	100	y	
0x1040			
0x1044			
0x1048			
0x1052			
0x1056			
0x1060			

```
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_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;
}
```

Apontadores e Vetores I

C/C++ permite manipulação de endereços via

- Indexação (v[4]) ou
- Aritmética de endereços (*(ap+4))

Apontadores e Vetores II

```
void imprime_vetor1(int v[], int n) {
    int i;
    for (i = 0; i < n; i++)
        cout << v[i] <<" ";
    cout << endl;
}

void imprime_vetor2(int* pv, int n) {
    int i;
    for (i = 0; i < n; i++)
        cout << pv[i] << " ";
    cout << endl;
}
```

Apontadores e Vetores III

```
void imprime_vetor3(int *pv, int n) {
    int i;
    for (i = 0; i < n; i++) {
        cout << *(pv + i) << " ";
    }
    cout << endl;
}
```

Apontadores e Vetores IV

```
int main() {
  int v[] = {10, 20, 30, 40, 50};

  imprime_vetor1(v, 4);
  imprime_vetor2(v, 4);
  imprime_vetor3(v, 4);

  return 0;
}
```

Mostra na tela
10 20 30 40 50
10 20 30 40 50
10 20 30 40 50

Vetores de apontadores I

```
int *vet_ap[5];
char *vet_cad[5];
```

- São vetores semelhantes aos vetores de tipos simples

Alocação dinâmica de memória I

- Aloca um bloco consecutivo de bytes na memória e retorna o endereço deste bloco;
- Permite escrever programas mais flexíveis.
- É utilizado o comando `new`
- Um ponteiro nulo (ptrnull) é um valor especial que podemos atribuir a um ponteiro para indicar que ele não aponta para lugar algum.

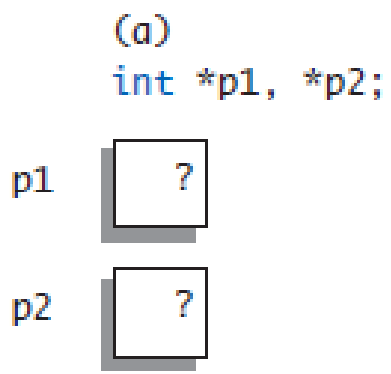
Alocação dinâmica de memória II

- O operador `new` permite criar uma variável dinâmica de um tipo específico e retorna o endereço da nova variável criada

```
int *n;
n = new int(17); // inicializa *n em 17
```

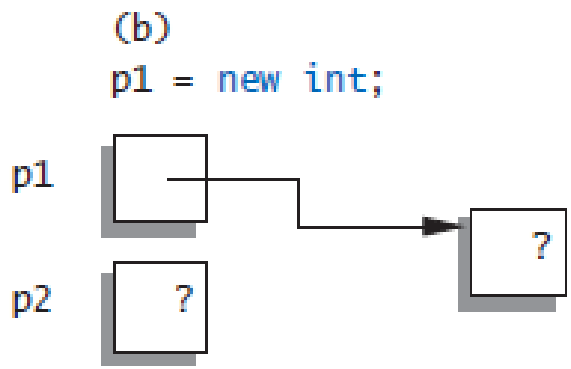
Alocação dinâmica de memória III

```
int main(){
→   int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p1 = p2;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p1 == " << *p2 << endl;
    delete p1;
    return 0;
}
```



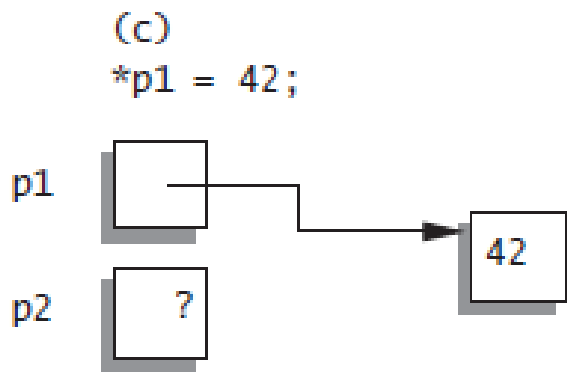
Alocação dinâmica de memória IV

```
int main(){
    int *p1, *p2;

→   p1 = new int;
    *p1 = 42;
    p1 = p2;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p1 == " << *p2 << endl;
    delete p1;
    return 0;
}
```



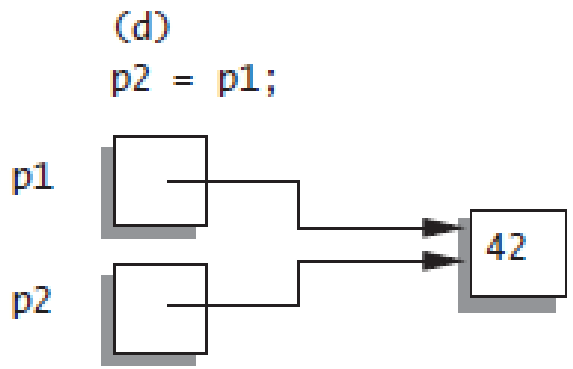
Alocação dinâmica de memória V

```
int main(){
    int *p1, *p2;

    p1 = new int;
→   *p1 = 42;
    p1 = p2;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p1 == " << *p2 << endl;
    delete p1;
    return 0;
}
```



Alocação dinâmica de memória VI

```
int main(){
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
→   p1 = p2;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p1 == " << *p2 << endl;
    delete p1;
    return 0;
}
```



Alocação dinâmica de memória VII

Usando as funções definidas nos slides anteriores

```
int main() {
  int *a = ptrnull;
  a = new int[6];
  for(int i = 0; i < 6; i++)
      a[i] = i;

  imprime_vetor2(a, 5);
  return 0;
}
```

Mostra na tela

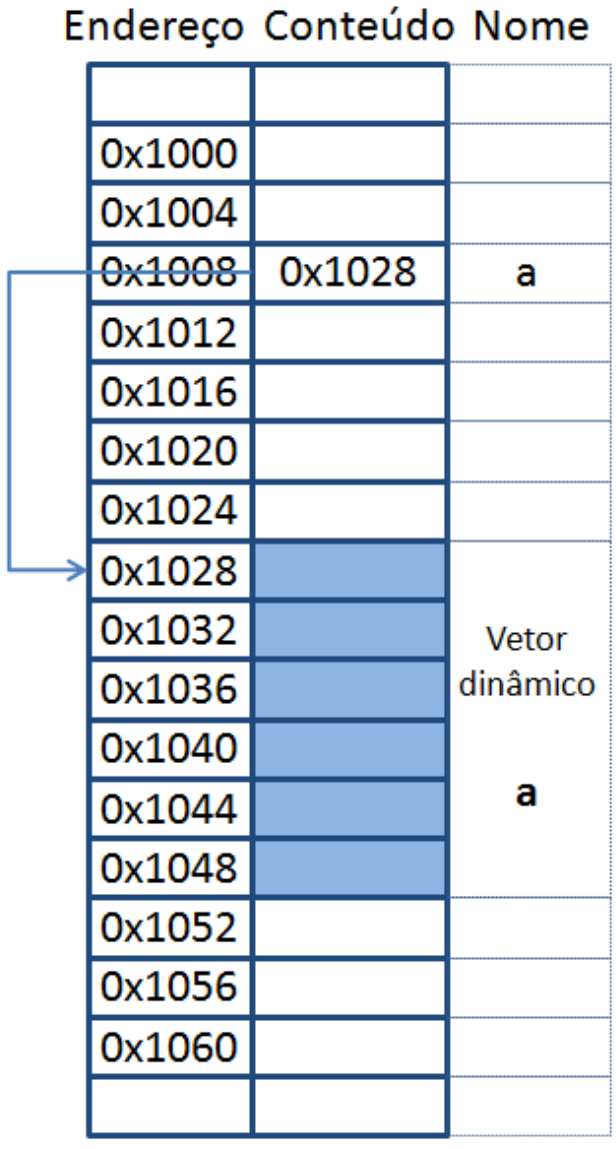
0 1 2 3 4 5

Alocação dinâmica de memória VIII

Endereço	Conteúdo	Nome
0x1000		
0x1004		
0x1008	0x0000	a
0x1012		
0x1016		
0x1020		
0x1024		
0x1028		
0x1032		
0x1036		
0x1040		
0x1044		
0x1048		
0x1052		
0x1056		
0x1060		

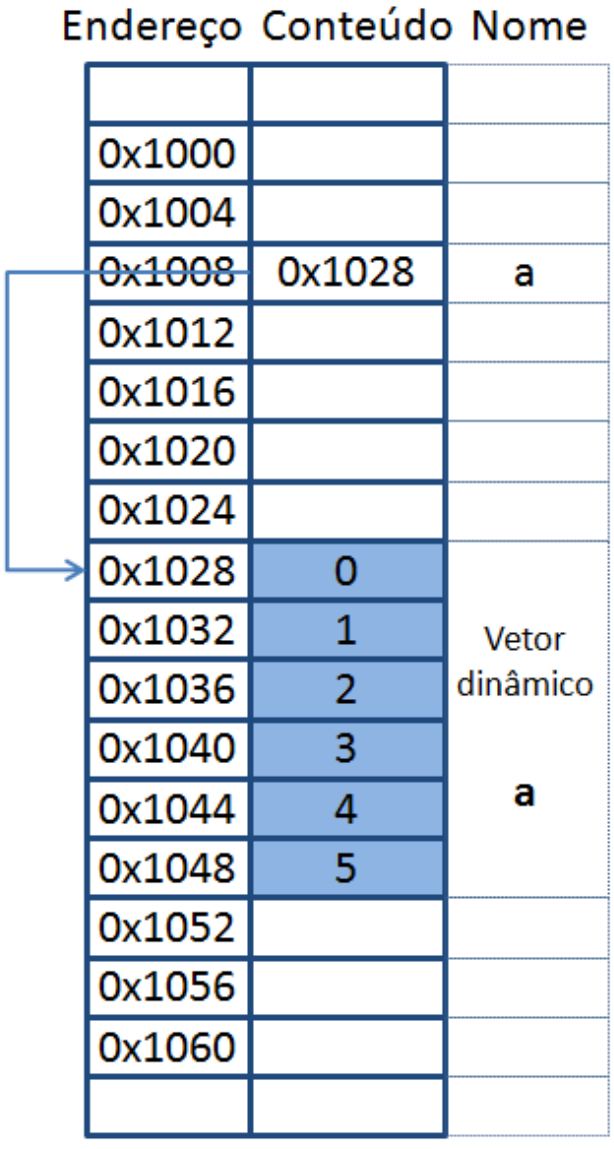
```
int main() {
  int *a = NULL;
  a = (int*) malloc (6 * sizeof(int));
  for(int i = 0; i < 6; i++)
     a[i] = i;

  imprime_vetor2(a, 5);

  return 0;
}
```

Alocação dinâmica de memória IX




```
int main() {
  int *a = NULL;
  a = (int*) malloc (6 * sizeof(int));
  for(int i = 0; i < 6; i++)
     a[i] = i;

  imprime_vetor2(a, 5);

  return 0;
}
```

Alocação dinâmica de memória X




```
int main() {
  int *a = NULL;
  a = (int*) malloc (6 * sizeof(int));
  for(int i = 0; i < 6; i++)
     a[i] = i;

  imprime_vetor2(a, 5);

  return 0;
}
```

Liberação de memoria I

- Libera o uso de um bloco de memória
- O comando a ser utilizado é: `delete`

Liberação de memoria II

No exemplo anterior faltou ser liberada a memoria alocada

```
int main() {
  int *a = nullptr;
  a = new int[6];
  for(int i = 0; i < 6; i++)
      a[i] = i;

  imprime_vetor2(a, 5);

  delete[] a;
  return 0;
}
```

Mostra na tela
0 1 2 3 4 5

Liberação de memoria III

- Na alocação dinâmica temos que verificar que a alocação foi feita com sucesso.
  - Usando exceções (método por *default*)

```
ptr = new int[5]; // se falhar, e lancada uma bad_alloc (exception)
```

  - Não permitindo o uso da excepção (*nothrow*)

```
ptr = new (std::nothrow) int[5]; // se falhar, retorna um ponteiro nulo (exception)
```

Liberação de memoria IV

```
#include <iostream>        // std::cout
#include <new>             // std::bad_alloc

int main () {
  try
  {
    int* myarray= new int[10000];
  }
  catch (std::bad_alloc& ba)
  {
    std::cerr << "Erro de memoria: " << ba.what() << '\n';
  }
  delete[] myarray;
  return 0;
}
```

Output Erro de memoria: bad allocation

Liberação de memoria V

```
#include <iostream>       // std::cout
#include <new>            // std::nothrow

int main () {
  std::cout << "Tentando alocar memoria";
  char* p = new (std::nothrow) char [1048576];
  if (p==0) std::cout << "Failed!\n";
  else {
    std::cout << "Succeeded!\n";
    delete[] p;
  }
  return 0;
}
```

Liberação de memoria VI

Crie uma função chamada `inverte_vetor`, que receba um vetor $V1$ e gera um novo vetor com os dados na ordem inversa

Liberação de memoria VII

```
void Inserir(int *V, int n);
int* inverte_vetor(int *V, int n);
void Mostrar(int *V, int n);
int main()
{
    int *V1 = nullptr, *V2 = nullptr, n;
    cout << "\n Digite o tamanho: ";
    cin >> n;
    V1 = new int[n];
    Inserir(V1, n);
    V2 = inverte_vetor(V1, n);
    cout << "\n Vetor original ";
    Mostrar(V1, n);
    cout << "\n Vetor invertido ";
    Mostrar(V1, n);
    delete[] V1;
    delete[] V2;
}
```

Liberação de memoria VIII

```
void Inserir(int *V, int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        cout << "\n Digite um valor: ";
        cin >> V[i];
    }
}
```

Liberação de memoria IX

```
int* inverte_vetor(int* V, int n)
{
    int i, j, *M = nullptr;
    M = new int[n];
    for (i = 0, j = n-1; i < n; i++, j--)
    {
        M[i] = V[j];
    }
    return M;
}
```

Liberação de memoria X

```
void Mostrar(int* V, int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        cout << V[i] << " ";
    }
    cout << endl;
}
```

Exercicios em Propostos I

1. Tente descobrir qual será o valor exibido pelo programa a seguir. Depois, digite e execute o programa para ver se você acertou. Explique o resultado observado.

```
#include <stdio.h>
int main() {
    int v, *p;
    v = 4;
    p = &v;
    *p = 2 * *p;
    cout << "\nValor: %" <<  v;
    return 0;
}
```

2. Crie um procedimento dobro(), que recebe dois parâmetros reais e devolve no segundo deles o dobro do valor do primeiro.

FIM