C++

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP

Introdução I

- C++ é uma extensão a linguagem C
- Desenvolvida por Bjarne Stroustrup na década de 80, nos laboratorios Bell
- Baseada em C e Simula67
- A maioria dos sistemas operacionais de hoje são escritos em C/C++

Introdução II

- Programas em C++ consistem de peças: classes e funções
- Podemos criar cada peça nós mesmos
  - No entanto, a maioria dos programadores tiram vantagem das ricas coleções de classes e funções da **Biblioteca Padrão C++ (C++ Standard Library)**
- Existem duas partes a serem apredidas
  - A linguagem C++ em si;
  - A utilização das classes e funções da biblioteca padrão C++

Introdução III

- **Vantagem:** ao criar nossas próprias classes e funções, nós conhecemos exatamente como funcionam
- **Desvantagem:** tempo gasto em codificar e a complexidade do projeto
- Evitar reinventar a roda!
  - Usar as peças existentes sempre que possível: **Reuso de Software**

Introdução IV

- **Problema da programação estruturada:** as unidades geradas não refletem entidades do mundo real com precisão
- Com a orientação a objetos, se o projeto é bem feito, há uma grande tendência a reutilização
- A linguagem C++ suporta todas as instruções e estruturas da linguagem C
- É possível programar em C puro e ser interpretado como C++

Estrutura Básica de um programa C I

```
< diretivas do pré-processador >
< declarações globais >;
int main()
{
    < declarações locais >;  /* comentário */
    < instruções>;
    return 0;
}
< outras funções >
```

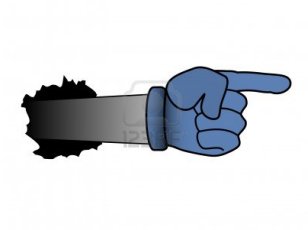
Estrutura Básica de um programa C II

```cpp
/* Prog. C++: Bom dia */
#include <iostream>
using namespace std;
int main()
{
    cout<<"Bom Dia!!";
    return 0;
}
```

```c
/* Prog. C: Bom dia */
#include <stdio.h>

int main()
{
    printf("Bom Dia!!");
    return 0;
}
```

- *main()* é única, determina o inicio do programa.
- O comando *return* informa ao sistema operacional se o programa funcionou corretamente ou não.

Regras para nomes de variáveis em C++ I



- Deve começar com uma letra (maiúscula ou minúscula) ou subscrito (_ *underscore*).
- **Nunca** pode começar com um número.
- Pode conter letras maiúsculas, minúsculas, número e subscrito
- Não pode-se utilizar `{ ( + - / \ ; . , ?` como parte do nome de uma variável.

Regras para nomes de variáveis em C++ II

- C/C++ são uma linguagem *case-sensitive*, ou seja, faz diferença entre nomes com letras maiúsculas e nomes com letras minúsculas: `Peso` e `peso` são diferentes.
- Costuma-se usar maiúsculas e minúsculas para separar palavras `PesoDoCarro`
- Identificadores devem ser únicos no mesmo escopo (não podem haver variáveis com mesmo identificador dentro do mesmo bloco).

Escrevendo o conteúdo de uma variável na tela em C++ I

- C++ usa o conceito de *streams* (fluxos) para executar operações de entrada e saída
- Uma *streams* é um objeto no qual um programa pode inserir dados ou do qual ele pode extrair dados.
- Para se utilizar *streams* , é necessário incluir a biblioteca `iostream`.

Escrevendo o conteúdo de uma variável na tela em C++ II

- Por default, a saída padrão envia dados para a tela e o objeto *stream* é identificado como `cout`.
- `cout` é usado em conjunto com o operador de inserção (<<).
- Exemplo: `cout << x << endl;`

Formatação de saída I

- *I/O manipulators* são a forma mais comum de controlar a formatação de saída. Usar a biblioteca `<iomanip>`

- Alguns métodos para manipular a formatação de saída:

Método	**Descrição**
`endl`	escreve uma nova linha
`setw(n)`	define o tamanho da saída. Só afeta ao elemento que vem a continuação
`width(n)`	igual que `setw(n)`
`left`	justifica à esquerda, so pode ser usado depois de `setw`
`right`	justifica à direita, so pode ser usado depois de `setw`

Formatação de saída II

- Exemplo 1:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    float x = 25.65749;
    cout << setw(10) << x;

    return 0;
}
```

imprime ␣␣25.65749

Formatação de saída III

Método	Descrição
`setfill(ch)`	usado depois de `setw`, preenche os espaços com o caracter definido em `ch`
`fixed`	mostra os decimais de um núm. real, por *default* são 6 decimais
`setprecision(n)`	define o número de decimais que serão mostrados. Deve ser usado junto com `fixed`. De não ser assim conta o número total de dígitos (inteiros e decimais).

Formatação de saída IV

- Exemplo 2:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    float x = 25.65749;
    cout << setfill('0')<< setw(11) << x;
    return 0;
}
```

imprime 00025.65749

Formatação de saída V

- Exemplo 3:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    float x = 49325.65749;
    cout << setprecision(4) << x;
    return 0;
}
```

imprime 4.933e+004

Formatação de saída VI

- Exemplo 4:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    float x = 49325.65749;
    cout << fixed << setprecision(4) << x;
    return 0;
}
```

imprime 49325.6575

Formatação de saída VII

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    const float A = 0.1;
    const float um  = 1.0;
    const float big  = 1234567890.0;
    const float B = 4567.87683;

    cout<<"A. "<<A<<", "<<um<<", "<<big<<endl;
    cout<<"B. "<<setprecision(5)<<B<<endl;
    cout<<"C. "<<fixed<<A <<", "<<um<<", "<<big<<endl;
    cout<<"D. "<<fixed<<setprecision(3)<<A<<", "
        <<um<<", "<<big<<endl;
    cout<<"E. "<<setprecision(20)<<A<<endl;
    cout<<"F. "<<setw(8)<<setfill('*')<<34<<45<<endl;
    cout<<"G. "<<setw(8)<<34<<setw(8)<<45<<endl;
    system("pause");
    return 0;
}
```

Formatação de saída VIII

Mostra na tela

A. 0.1, 1, 1.23457e+009
B. 4567.9
C. 0.10000, 1.00000, 1234567936.00000
D. 0.100, 1.000, 1234567936.000
E. 0.100000001490116120000
F. ******3445
G. ******34******45

A função cin (C++) I

- O operador >> sobrecarregado executa a entrada com streams em C++.
- O comando `cin` é utilizado para aquisição de dados
- Funções membro:

Método	**Descrição**
`get`	extrae caracteres
`getline`	lê caracteres até encontrar um determinado caracter
`ignore`	elimina caracteres do buffer até encontrar um determinado caracter
`clear`	reinicializa o flag de erro da entrada de dados
`gcount`	retorna o número de caracteres lidos

A função cin (C++) II

```
#include <iostream>
using namespace std;
int main(){
    int n;
    cout  << "Digite um número: ";
    cin >> n;
    cout << "O valor digitado foi " << n << endl;
    return 0;
}
```

A função cin (C++) III

Consistencia da entradas

```
#include <iostream>
using namespace std;
int main(){
    int num;
    do{
        cout << "Digite numero: ";
        cin >> num;
        if ( !cin.fail() ) break;
        cin.clear();
        cin.ignore(100, '\n');
    } while (true);
    return 0;
}
```

- O operador de **referência &** também é utilizado em C++
  - O operador é utilizado junto ao tipo, e não à variável
  - O termo referência é utilizado como um outro nome para uma variável já existente
  - Uma referência não é uma cópia de uma variável, é a mesma variável sob um nome diferente
  - Toda referência deve ser obrigatoriamente inicializada

```
#include <iostream>
using namespace std;
int main(){
    int n;
    int& a = n; // referência

    num = 15; // também altera a

    cout << n << endl;
    cout << a << endl;

    a = 0;   // também altera a

    cout << n << endl;
    cout << a << endl;
    return 0;
}
```

Programando em C++ III

- Um dos usos de referências é na passagem por referência
  - A função pode acessar as variáveis originais da função que a chamou;
  - Note que há uma inversão em relação à linguagem C
    - Na chamada da função, passamos apenas o nome da variável;
    - No cabeçalho da função, usamos o operador de referência junto ao tipo;
    - No corpo da função não é necessário tratamento especial.

Programando em C++ IV

```
#include <iostream>
using namespace std;
void troca(int& a, int& b){
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
int main(){
    int a = 10, b = 5
    cout << a << " " << b << endl;
    troca(a, b);
    cout << a << " " << b << endl;
    return 0;
}
```

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
00000001	
00000002	
00000003	
00000004	s
00000005	??
00000006	??
00000007	??
00000008	??
00000009	??
0000000A	??
0000000B	??
0000000C	??
0000000D	??
0000000C	??
0000000D	??

i

} c

```
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.

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 (nullptr) 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
```

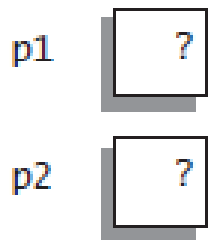
Alocação dinâmica de memória III

```
int main(){
→   int *p1, *p2;

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

Alocação dinâmica de memória IV

(a)

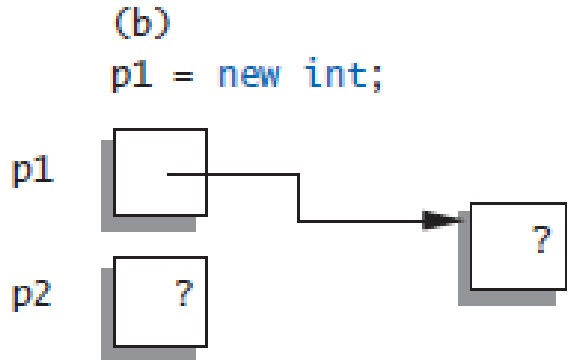
```
int *p1, *p2;
```



Alocação dinâmica de memória V

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

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

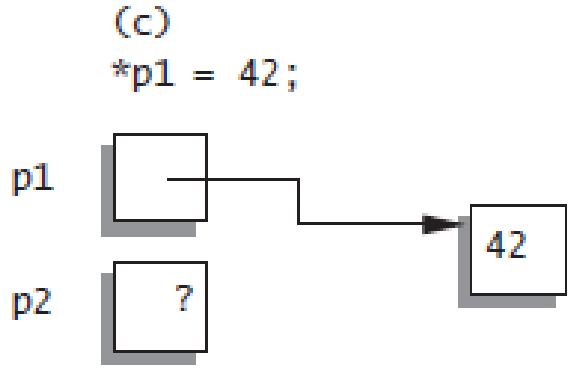
Alocação dinâmica de memória VI



Alocação dinâmica de memória VII

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

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

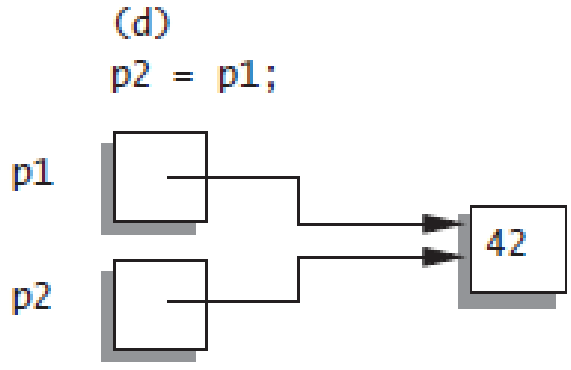
Alocação dinâmica de memória VIII



Alocação dinâmica de memória IX

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

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

Alocação dinâmica de memória X



Alocação dinâmica de memória XI

Usando as funções definidas nos slides anteriores

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

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

Mostra na tela

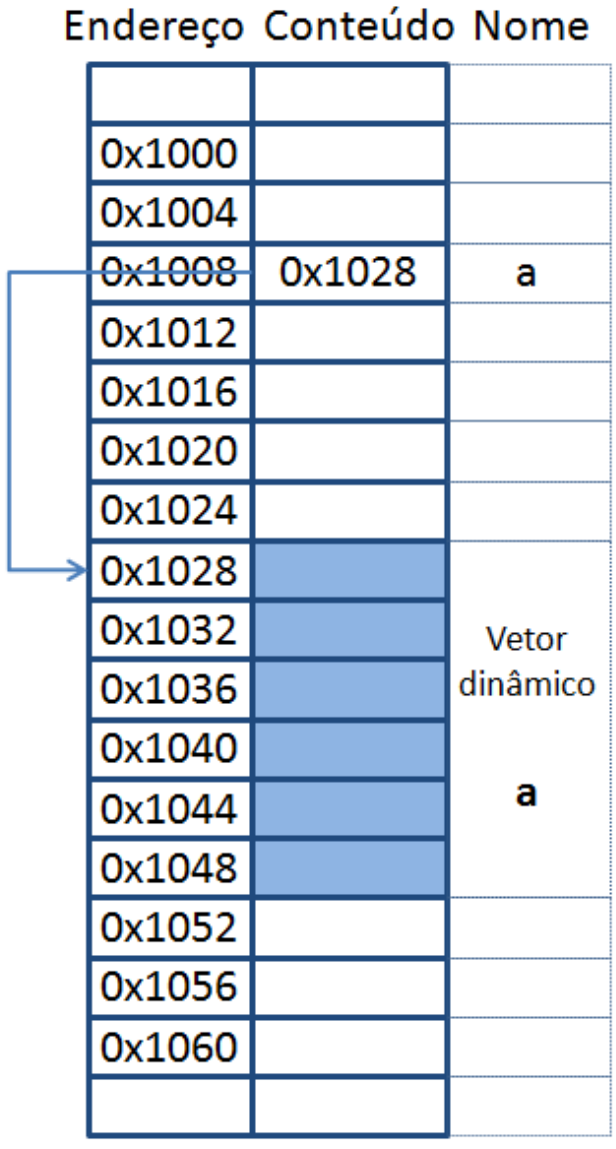
0 1 2 3 4 5

Alocação dinâmica de memória XII

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

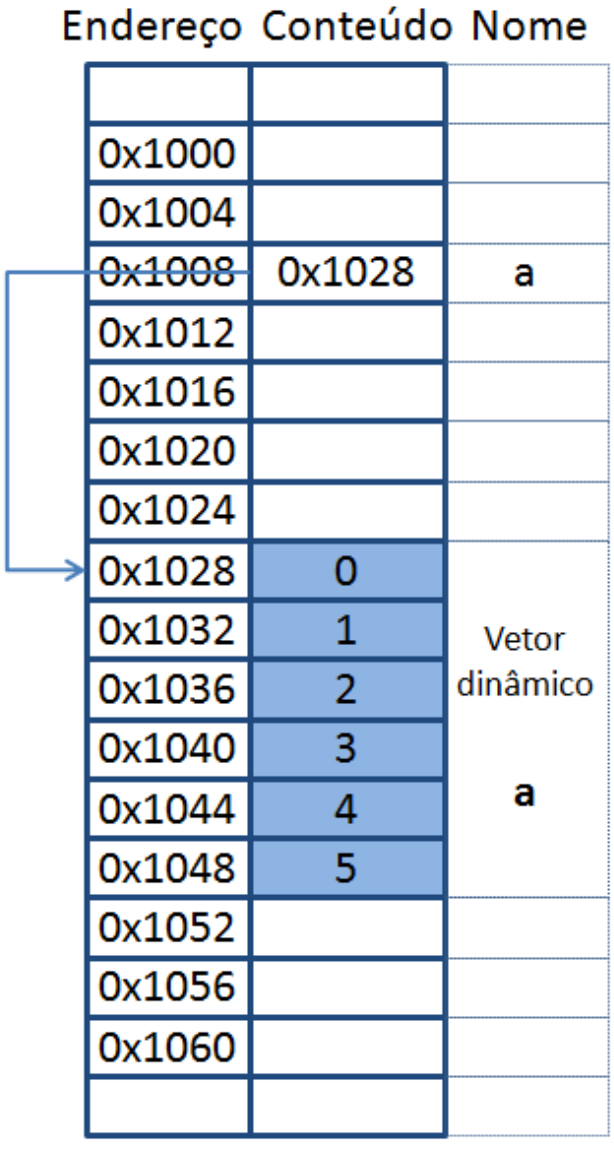
```
int main() {
  int *a = NULL;
  a = new int[6]
  for(int i = 0; i < 6; i++)
     a[i] = i;

  imprime_vetor2(a, 5);

  return 0;
}
```

Alocação dinâmica de memória XIII




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

  imprime_vetor2(a, 5);

  return 0;
}
```

Alocação dinâmica de memória XIV




```
int main() {
  int *a = NULL;
  a = new int[6];
  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;
}
```

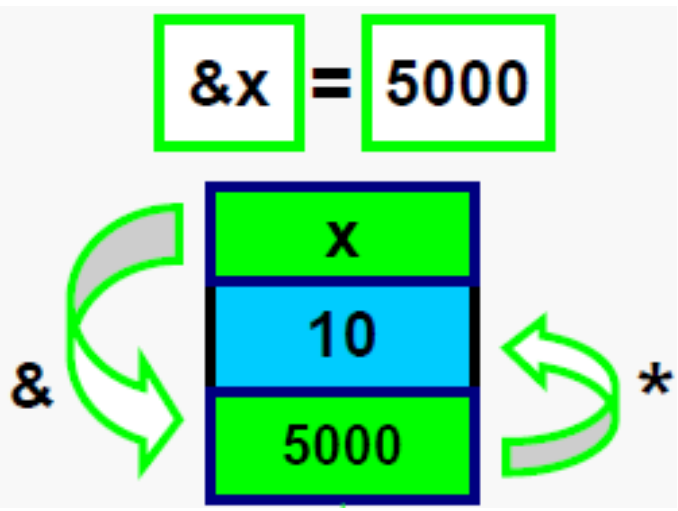
Output Erro de memoria: bad allocation

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;
}
```

Ponteiros I

```
int main()
{
    int x;
    x = 10
    cout << "Conteudo de x: " << x;
    cout << "Endereco de x: " << &x;
    return 0;
}
```

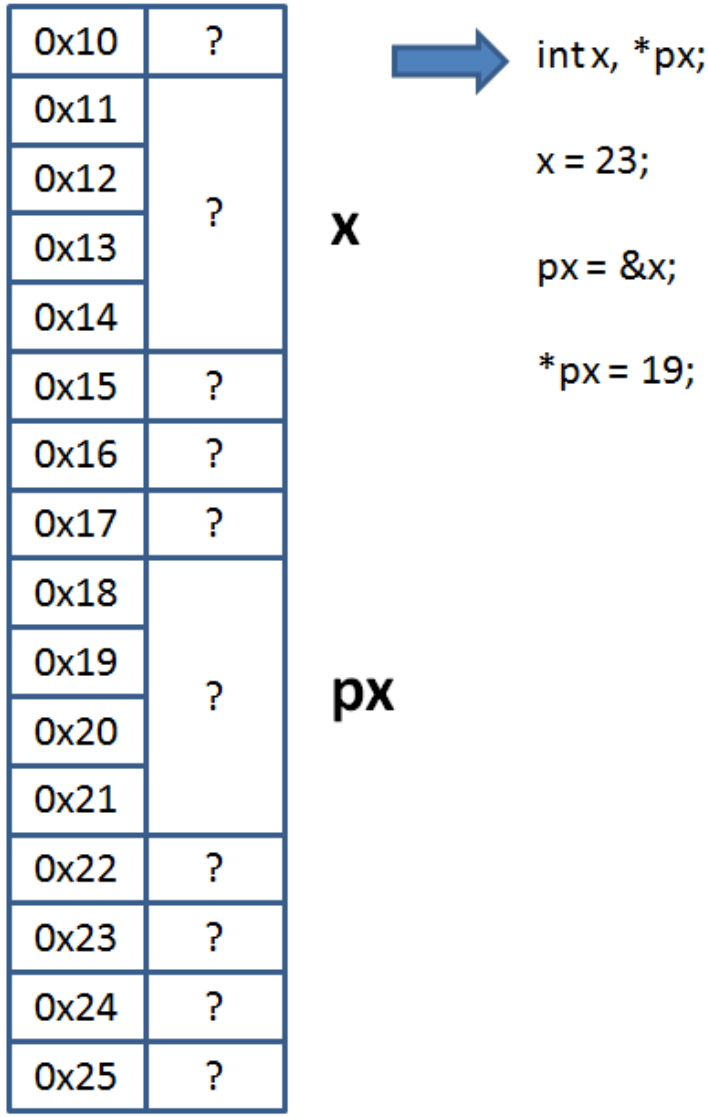


Ponteiros II

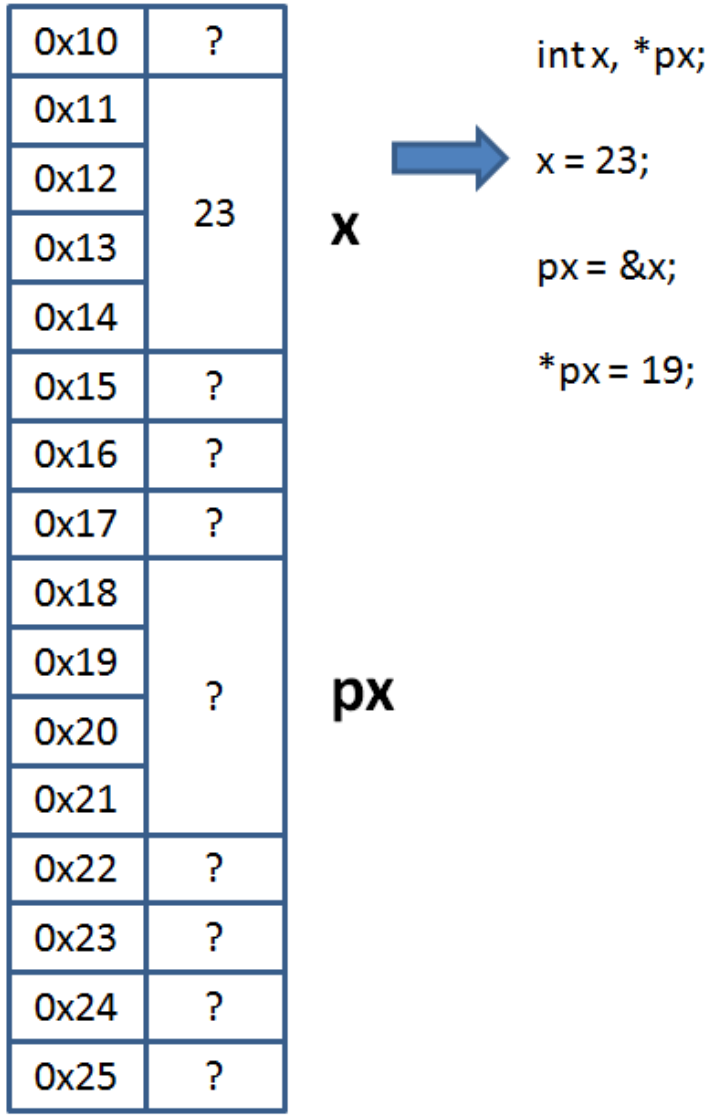
0x10	?
0x11	?
0x12	?
0x13	?
0x14	?
0x15	?
0x16	?
0x17	?
0x18	?
0x19	?
0x20	?
0x21	?
0x22	?
0x23	?
0x24	?
0x25	?

```
int x, *px;

x = 23;

px = &x;

*px = 19;
```

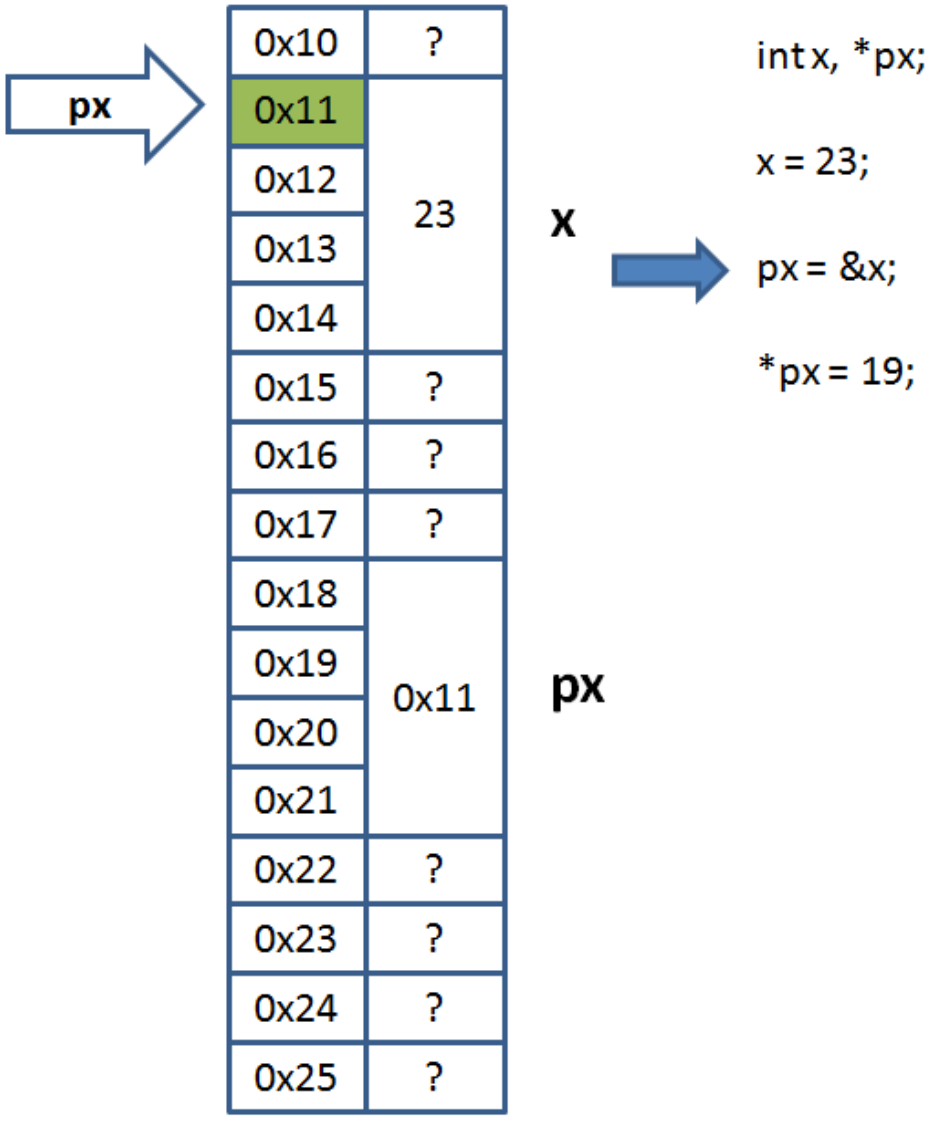
Ponteiros III



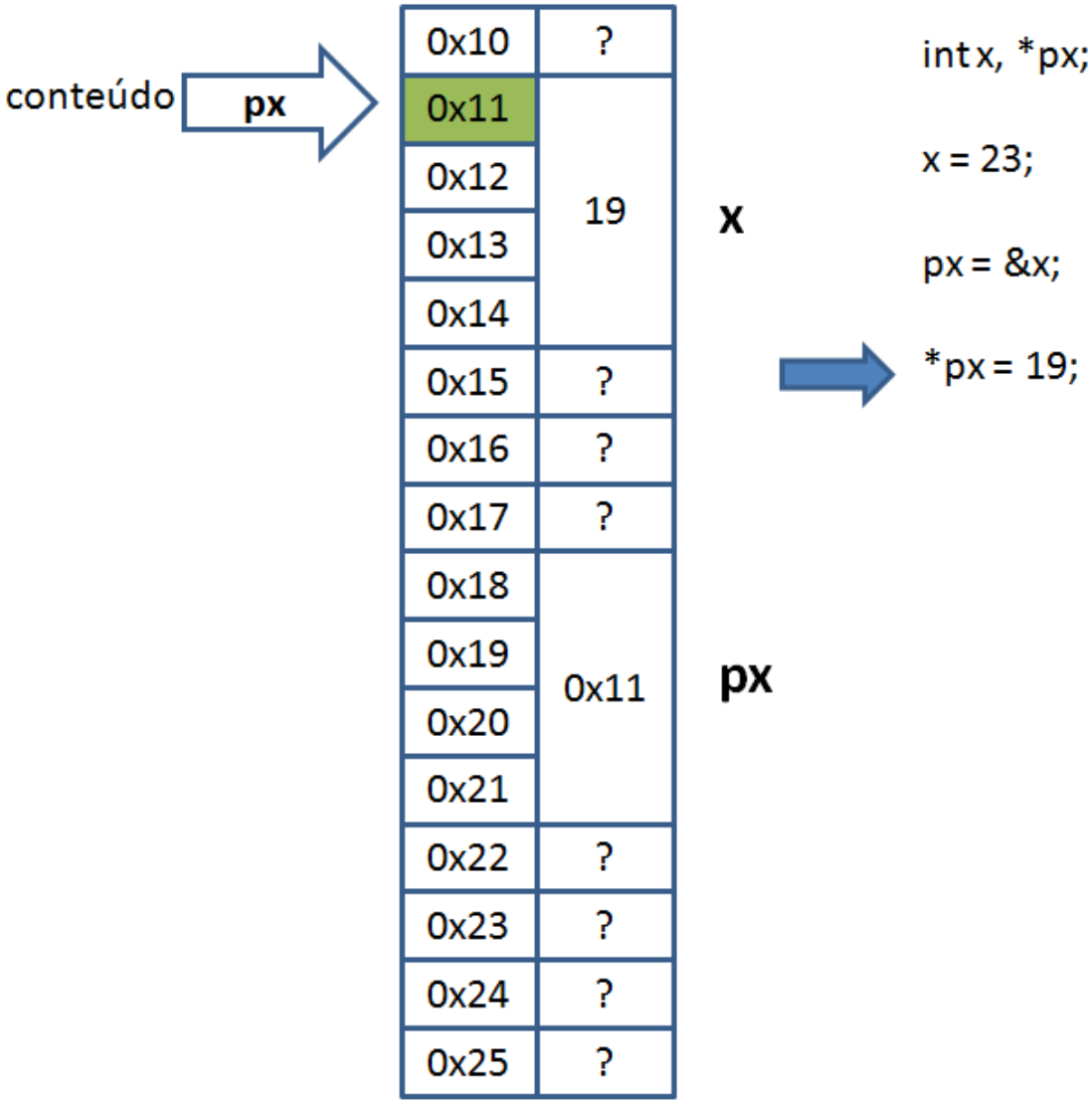
Ponteiros IV



Ponteiros V


px
0x10
?
0x11
0x12
0x13
0x14
23
x
0x15
?
0x16
?
0x17
?
0x18
0x19
0x20
0x21
0x11
px
0x22
?
0x23
?
0x24
?
0x25
?
int x, *px;
x = 23;
px = &x;
*px = 19;

Ponteiros VI



Ponteiros VII

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

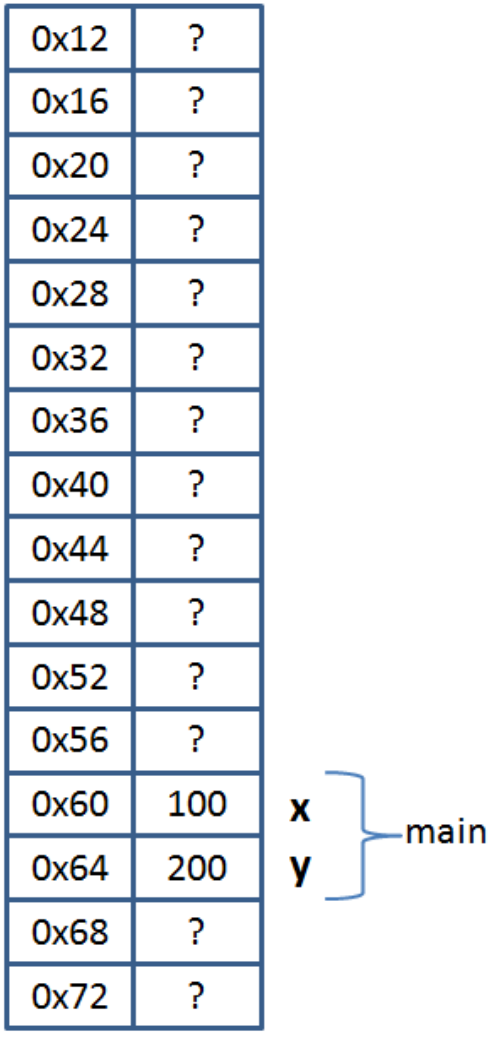
```
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 cout << "x = “<<x<<“ y = “<<y;
 troca(&x, &y);
 cout << "x = “<<x<<“ y = “<<y;
 return 0;
}
```

Ponteiros VIII

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

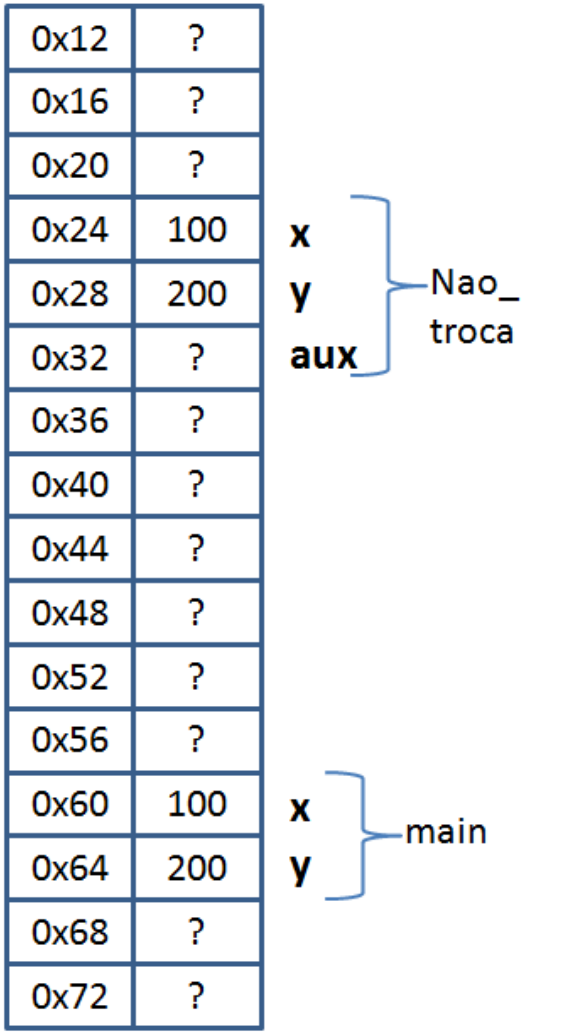
```
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 cout << "x = "<<x<<" y = "<<y;
 troca(&x, &y);
 cout << "x = "<<x<<" y = "<<y;
 return 0;
}
```

Ponteiros IX



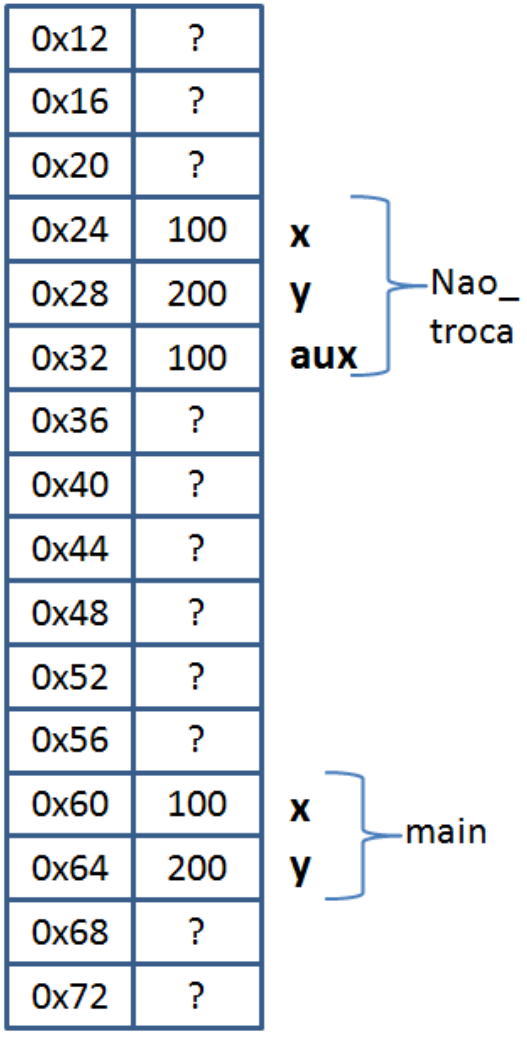

```
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 cout << "x = "<<x<<" y = "<<y;
 troca(&x, &y);
 cout << "x = "<<x<<" y = "<<y;
 return 0;
}
```

Ponteiros X



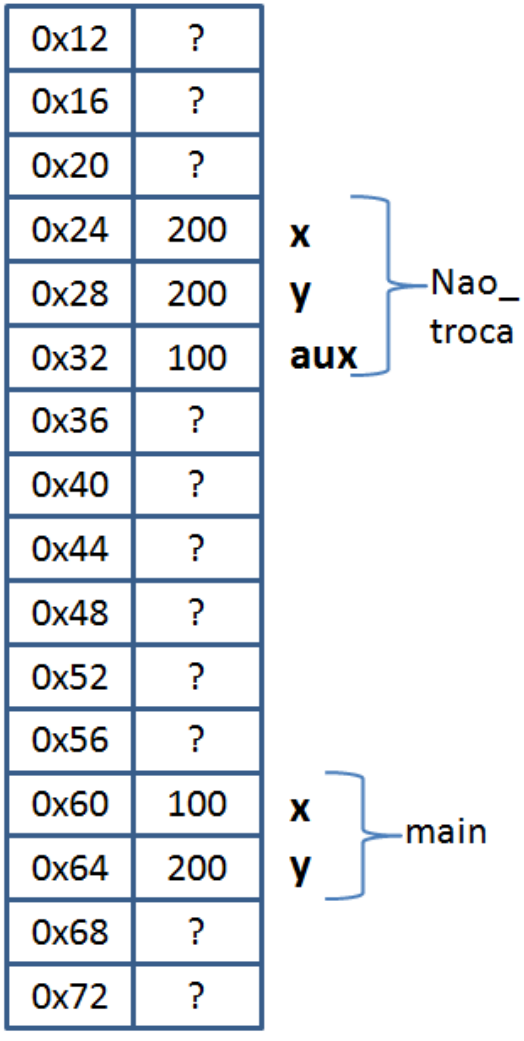

```
void troca(int *ap_x, int *ap_y) {
  int aux;
  aux = *ap_x;
  *ap_x = *ap_y;
  *ap_y = aux;
}
void nao_troca(int x, int y) {
  int aux;
  aux = x;
  x = y;
  y = aux;
}
int main() {
  int x = 100, y = 200;
  nao_troca(x, y);
  cout << "x = "<<x<<" y = "<<y;
  troca(&x, &y);
  cout << "x = "<<x<<" y = "<<y;
  return 0;
}
```

Ponteiros XI



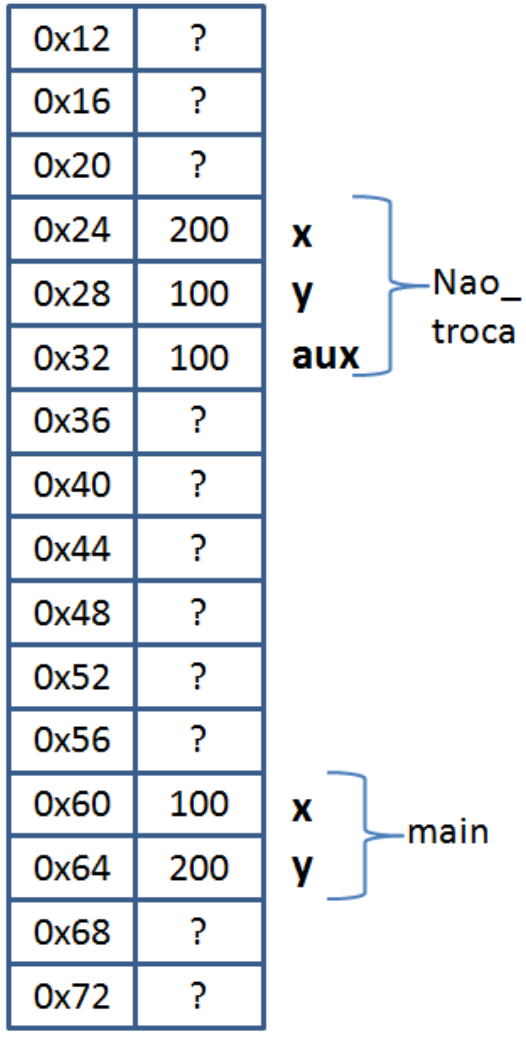

```
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 cout << "x = "<<x<<" y = "<<y;
 troca(&x, &y);
 cout << "x = "<<x<<" y = "<<y;
 return 0;
}
```

Ponteiros XII



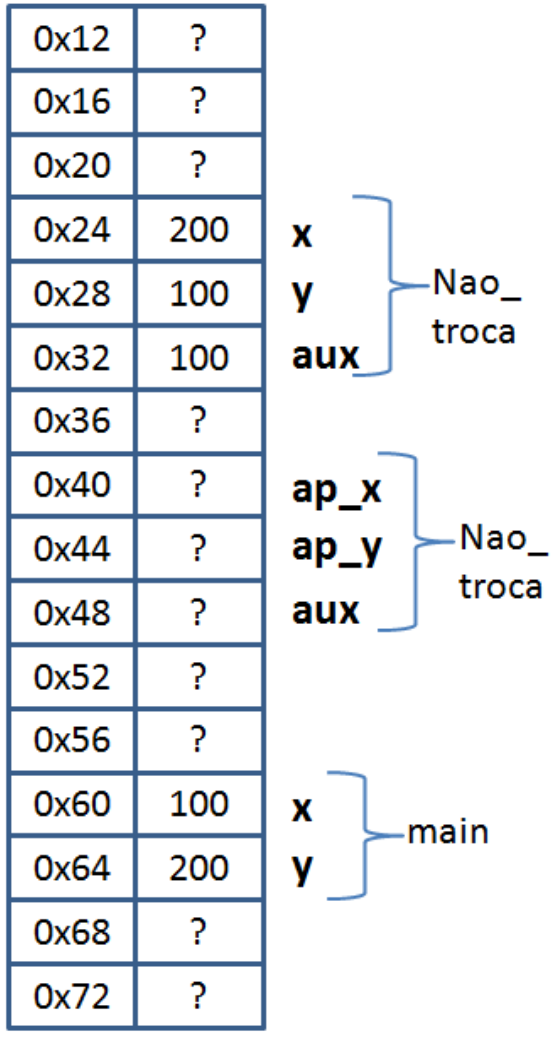

```
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 cout << "x = "<<x<<" y = "<<y;
 troca(&x, &y);
 cout << "x = "<<x<<" y = "<<y;
 return 0;
}
```

Ponteiros XIII



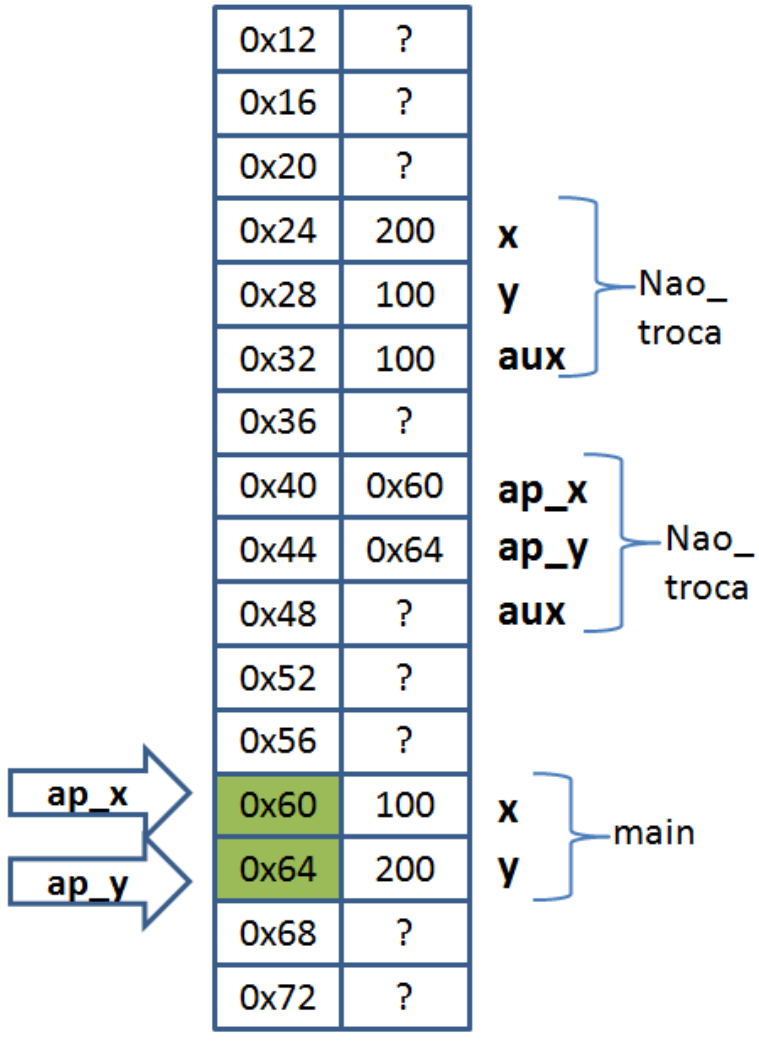

```
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 cout << "x = "<<x<<" y = "<<y;
 troca(&x, &y);
 cout << "x = "<<x<<" y = "<<y;
 return 0;
}
```

Ponteiros XIV



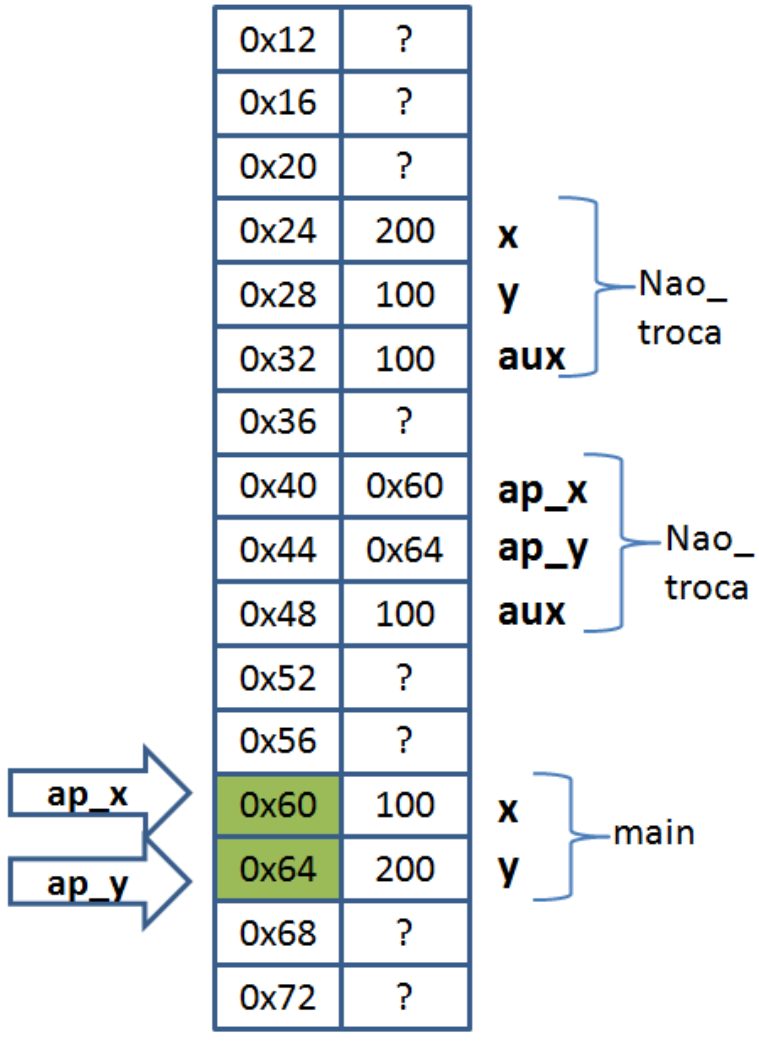

```
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 cout << "x = "<<x<<" y = "<<y;
 troca(&x, &y);
 cout << "x = "<<x<<" y = "<<y;
 return 0;
}
```

Ponteiros XV



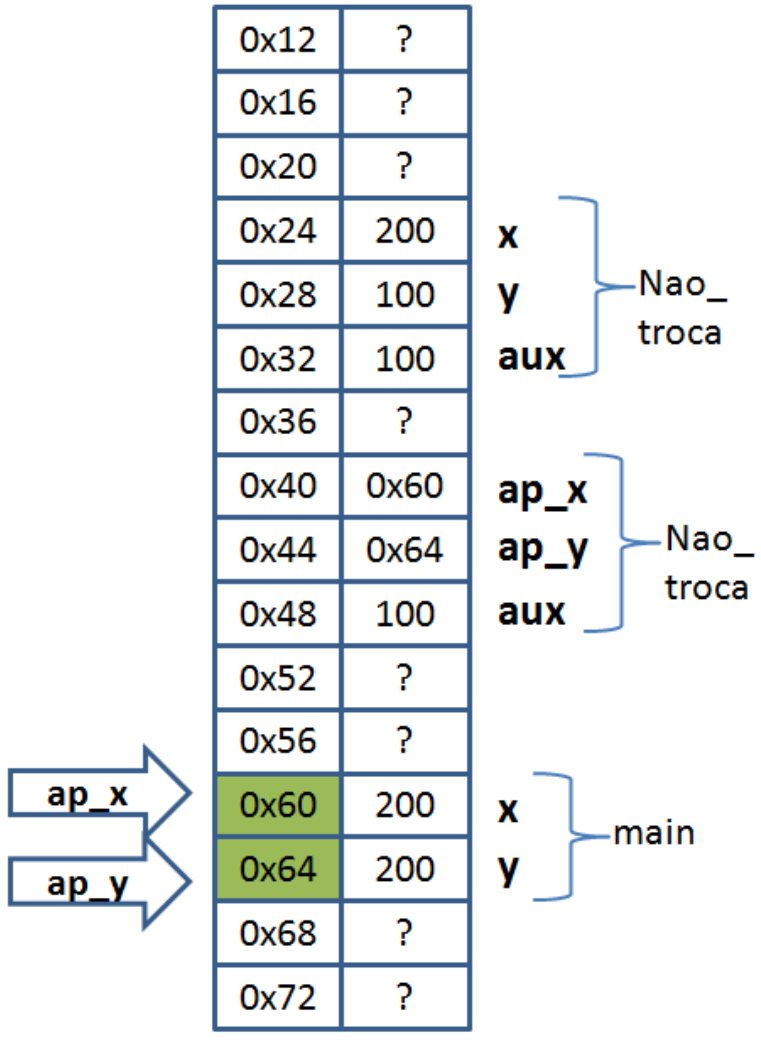

```
void troca(int *ap_x, int *ap_y) {
  int aux;
  aux = *ap_x;
  *ap_x = *ap_y;
  *ap_y = aux;
}
void nao_troca(int x, int y) {
  int aux;
  aux = x;
  x = y;
  y = aux;
}
int main() {
  int x = 100, y = 200;
  nao_troca(x, y);
  cout << "x = "<<x<<" y = "<<y;
  troca(&x, &y);
  cout << "x = "<<x<<" y = "<<y;
  return 0;
}
```

Ponteiros XVI



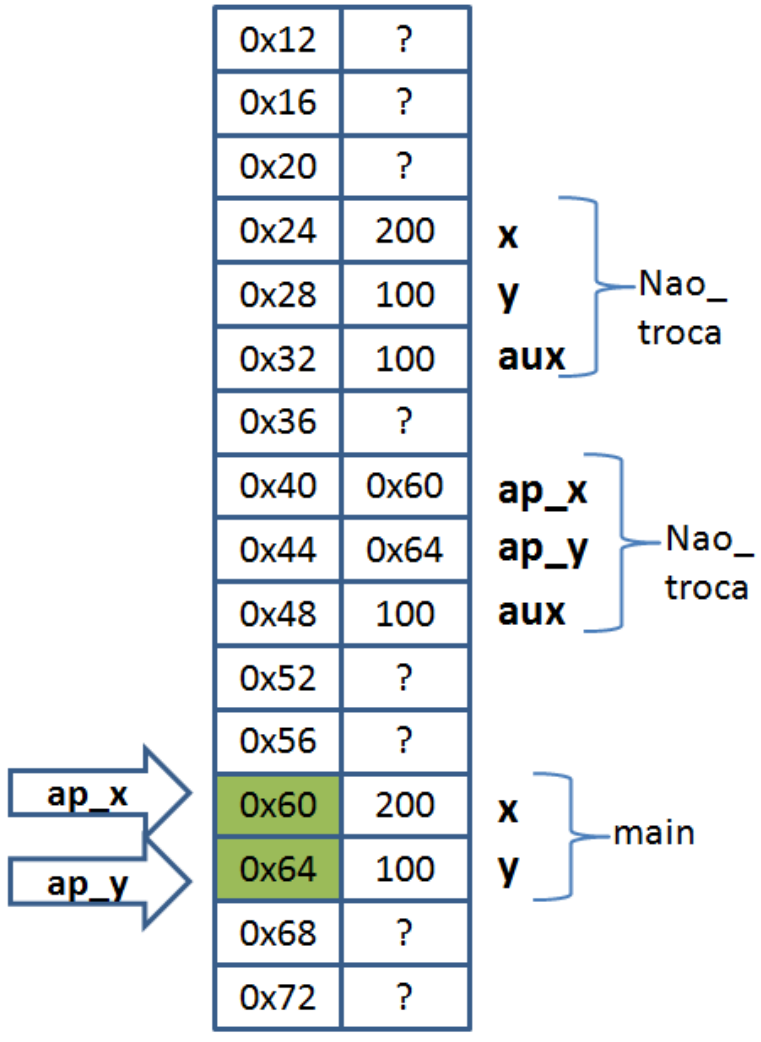

```
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 cout << "x = "<<x<<" y = "<<y;
 troca(&x, &y);
 cout << "x = "<<x<<" y = "<<y;
 return 0;
}
```

Ponteiros XVII




```
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);
  cout << "x = "<<x<<" y = "<<y;
  troca(&x, &y);
  cout << "x = "<<x<<" y = "<<y;
  return 0;
}
```

Ponteiros XVIII

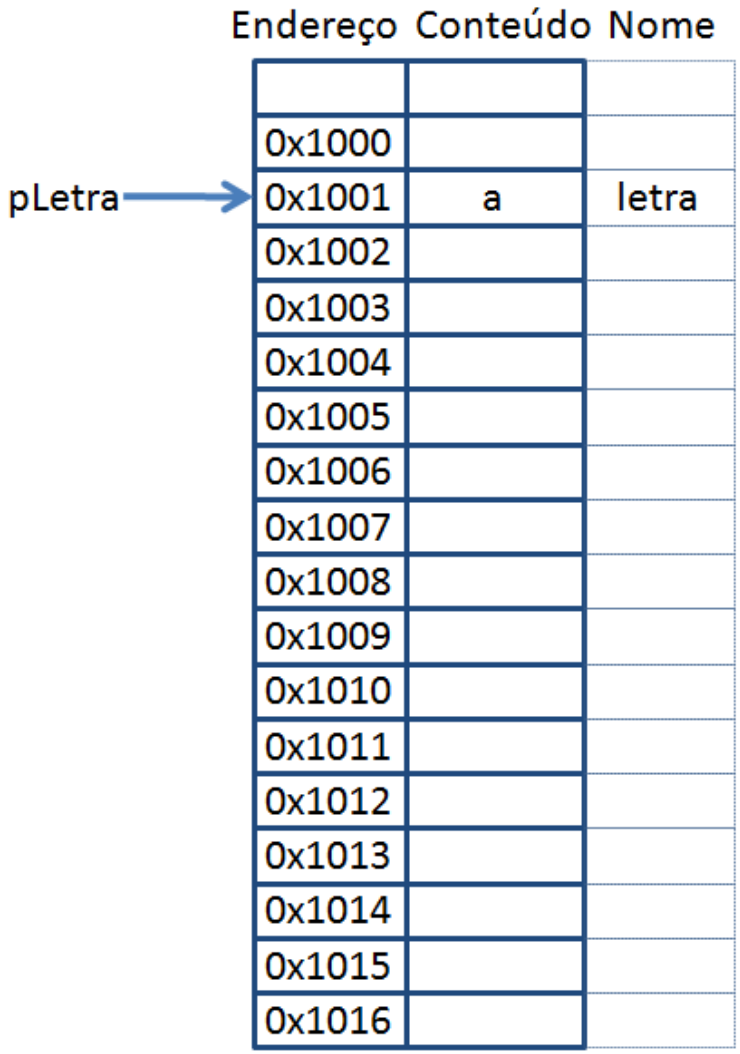



```
void troca(int *ap_x, int *ap_y) {
 int aux;
 aux = *ap_x;
 *ap_x = *ap_y;
 *ap_y = aux;
}
void nao_troca(int x, int y) {
 int aux;
 aux = x;
 x = y;
 y = aux;
}
int main() {
 int x = 100, y = 200;
 nao_troca(x, y);
 cout << "x = "<<x<<" y = "<<y;
 troca(&x, &y);
 cout << "x = "<<x<<" y = "<<y;
 return 0;
}
```

Aritmética de Ponteiros I

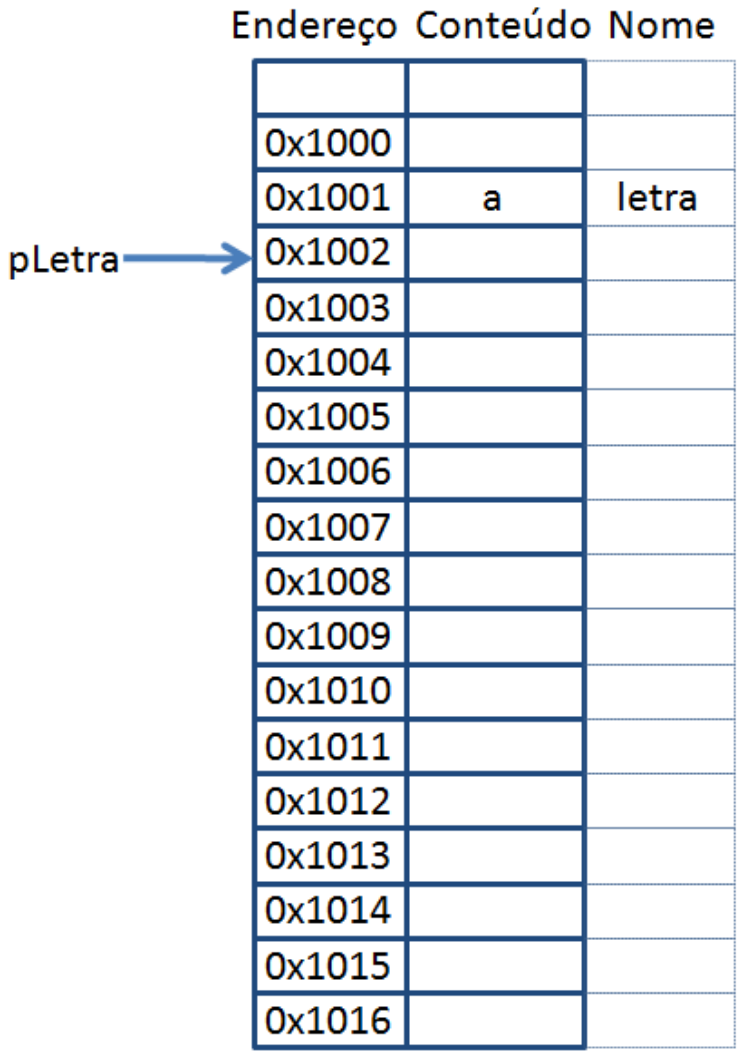
- Uma variável do tipo ponteiro está sempre associada a um tipo
- Um ponteiro para um dado tipo t endereça o número de bytes que esse tipo t ocupa na memória, i.e., endereça **sizeof(t)** bytes.
- Se um ponteiro para uma variável do tipo t for incrementada através do operador $++$, automaticamente este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

Aritmética de Ponteiros II



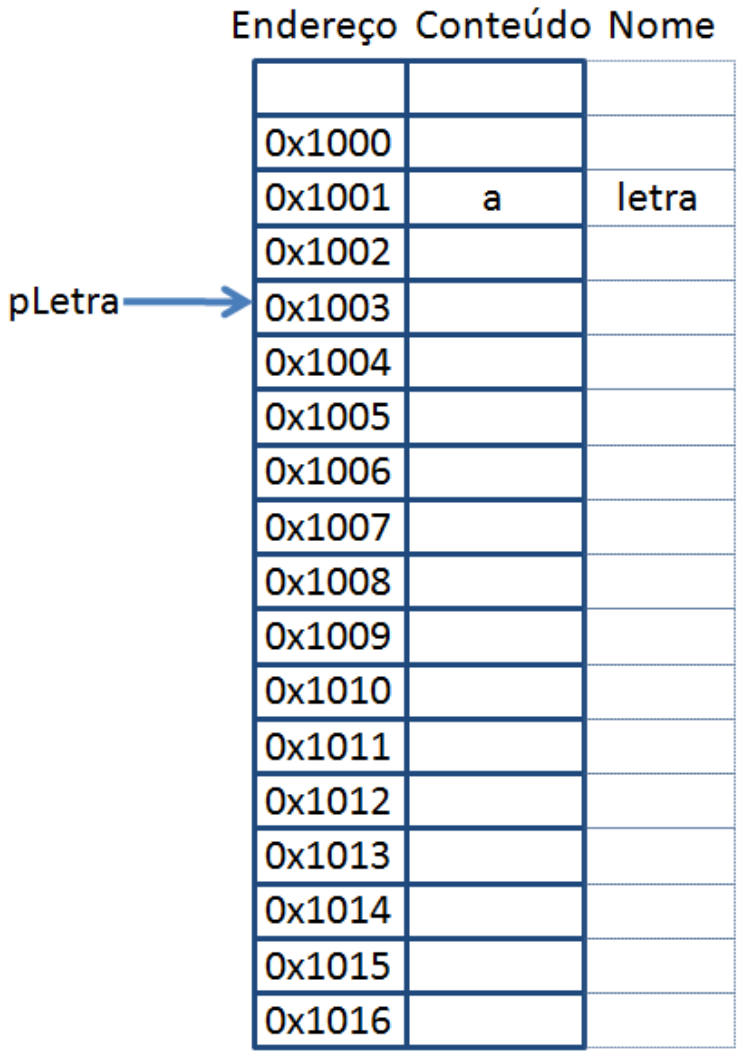

```
int main() {
  char* pChar, letra = 'a';
   pLetra = &letra;
   ...
   pLetra++;
  return 0;
}
```

Aritmética de Ponteiros III



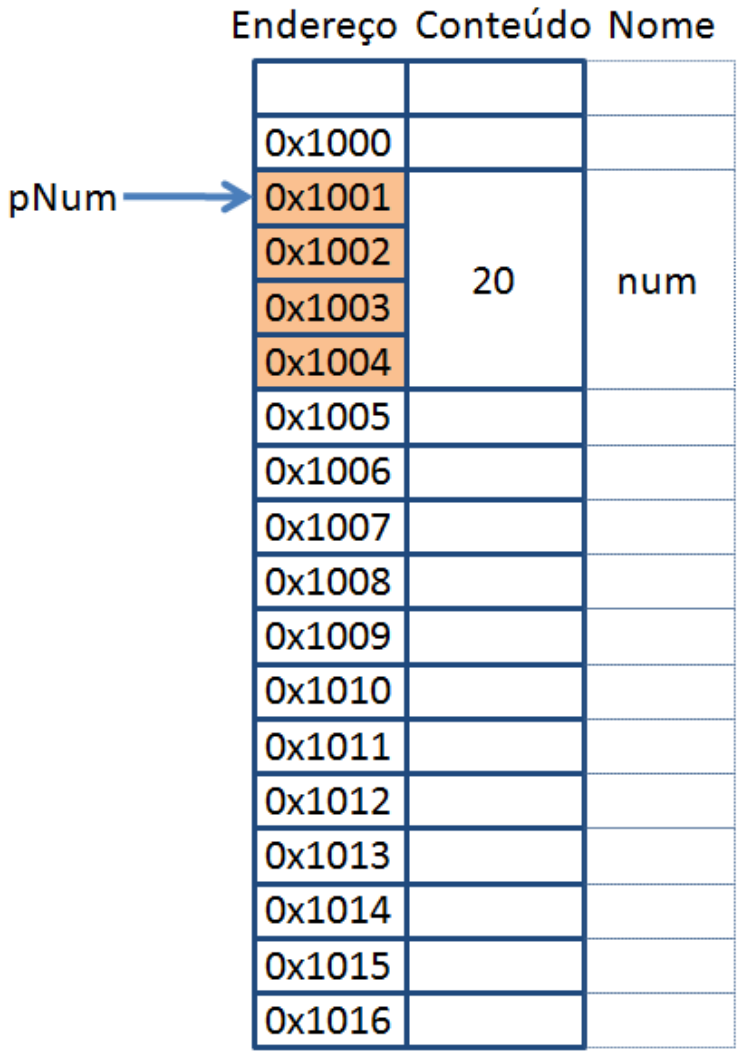

```
int main() {
  char* pChar, letra = 'a';
   pLetra = &letra;
   ...
   pLetra++;
   pLetra++;
  return 0;
}
```

Aritmética de Ponteiros IV



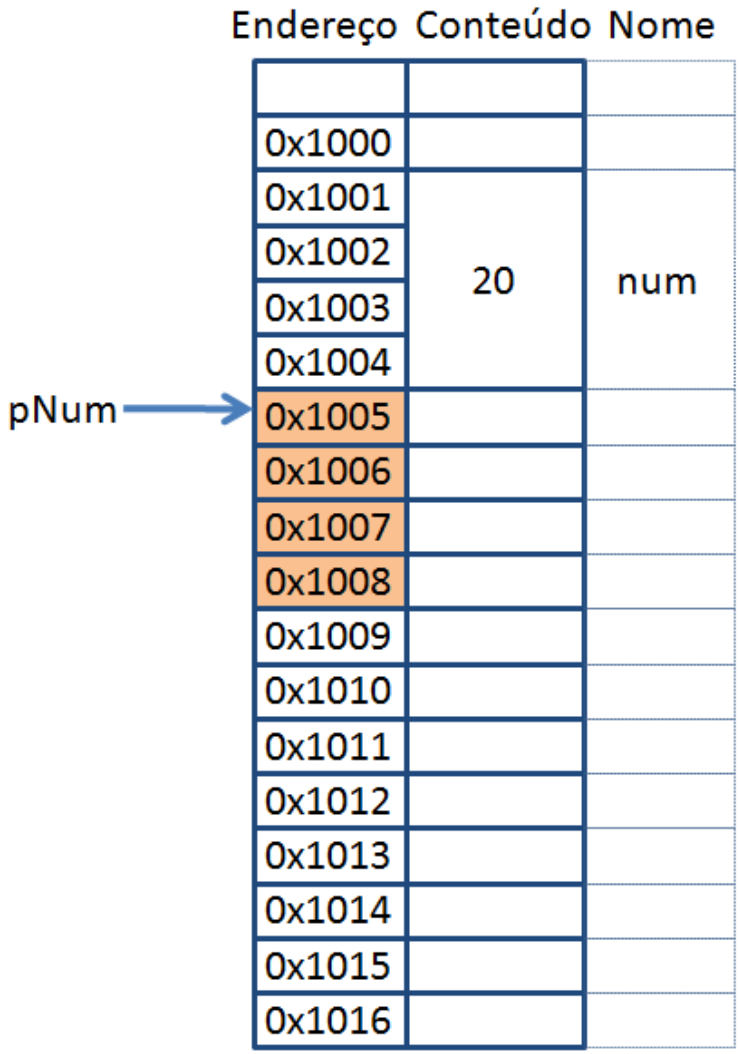

```
int main() {
  char* pChar, letra = 'a';
   pLetra = &letra;
   ...
   pLetra++;
   pLetra++;
  return 0;
}
```

Aritmética de Ponteiros V



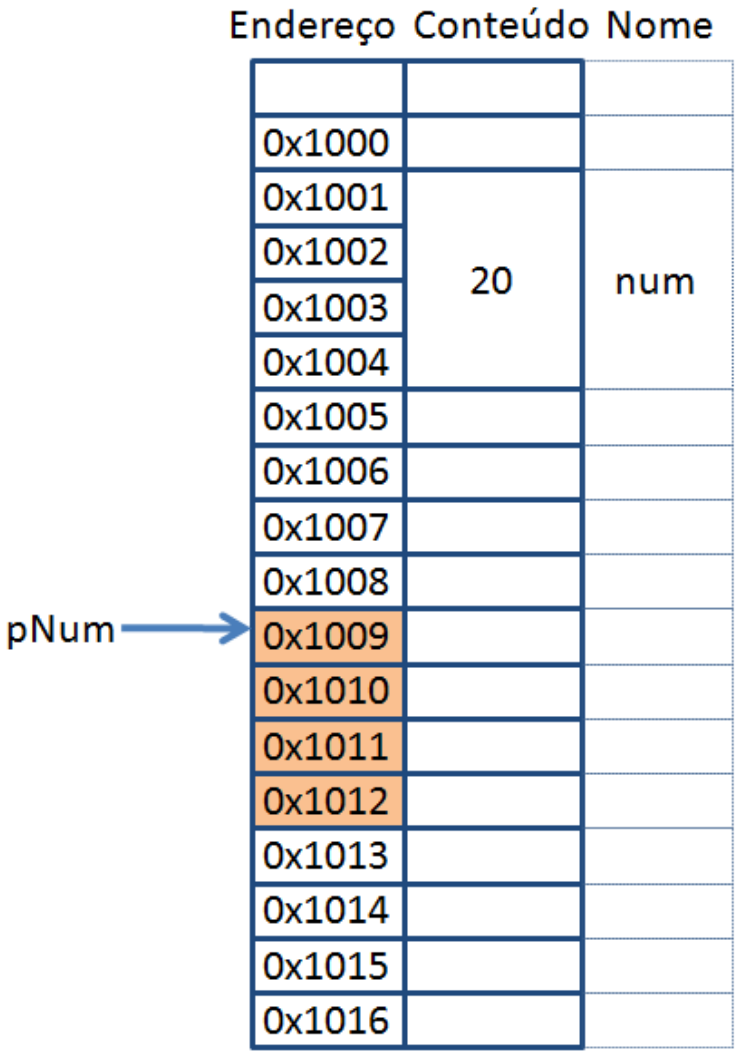

```
int main() {
  int* pNum, num = 20;
   pNum = &Num;
   ...
   pNum++;
  return 0;
}
```

Aritmética de Ponteiros VI



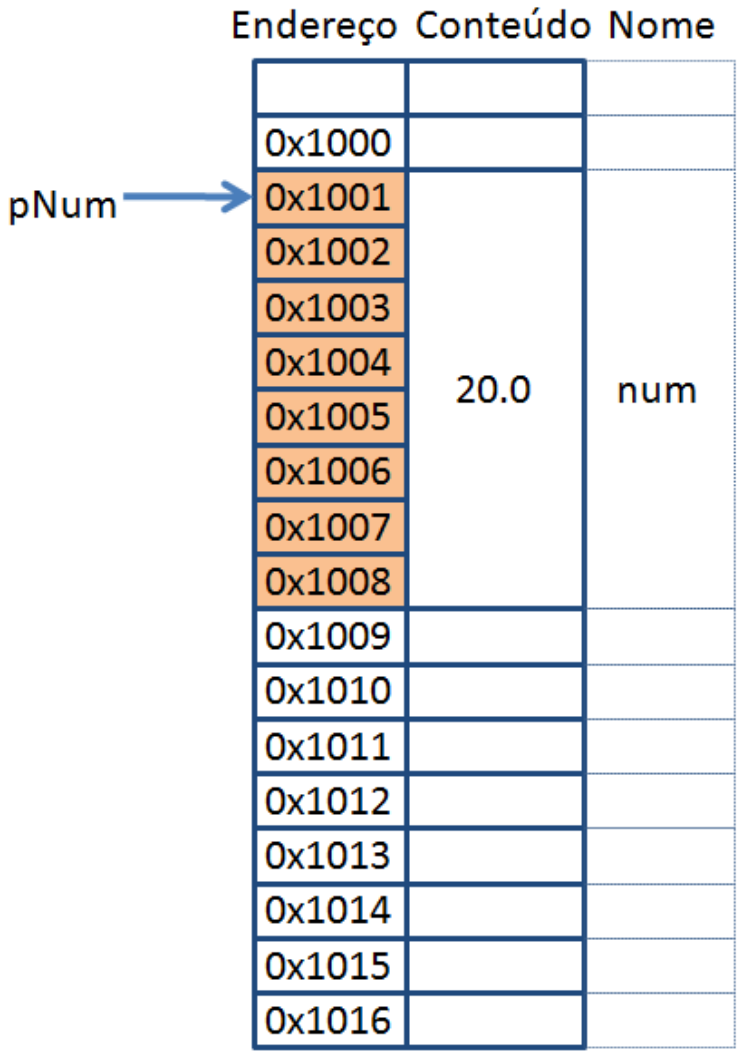

```
int main() {
  int* pNum, num = 20;
   pNum = &Num;
   ...
   pNum++;
  return 0;
}
```

Aritmética de Ponteiros VII



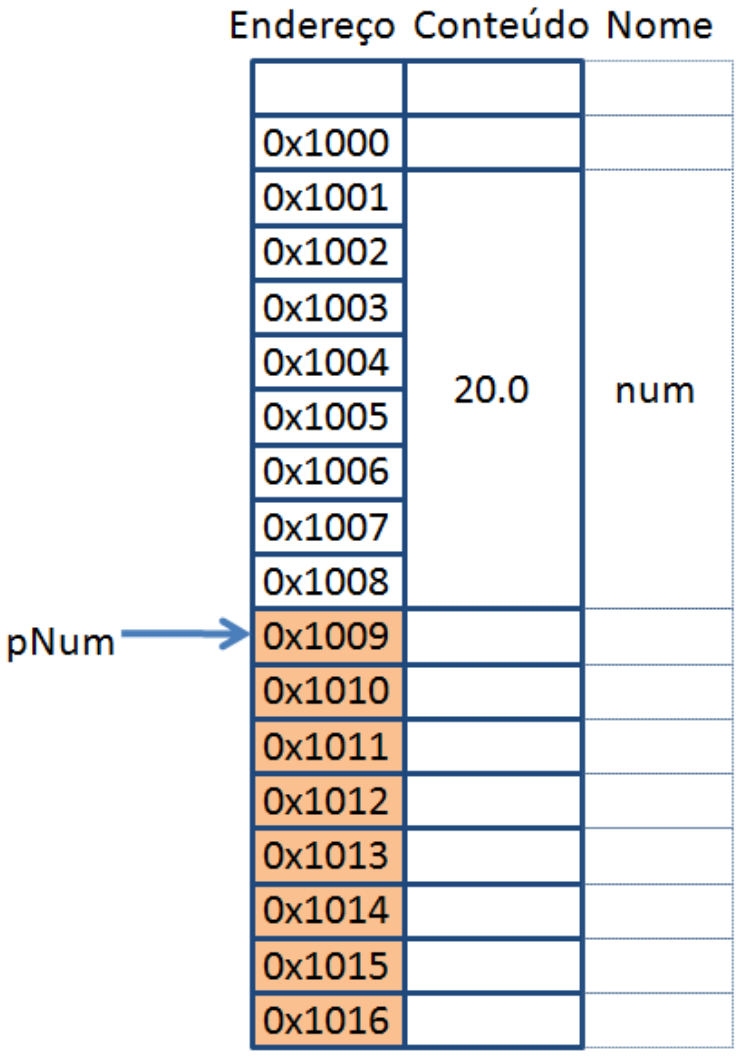

```
int main() {
  int* pNum, num = 20;
   pNum = &Num;
   ...
   pNum++;
   pNum++

  return 0;
}
```

Aritmética de Ponteiros VIII




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

Aritmética de Ponteiros IX




```
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

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

Alocação Dinâmica de Memória I

- C++ possui dois operadores que substituem a finalidade das funções *malloc* e *calloc*
  - *new* e *delete*
  - Mais adequados
  - Não retornam ponteiros *void*
  - Retornam ponteiros do tipo adequado ao necessário
  - Não é necessário utilizar *cast*

Alocação Dinâmica de Memória II

- O operador *new* solicita memória ao sistema operacional
  - Retornando um ponteito para o primeiro *byte* da memória alocada
- O operador *delete* libera a memória alocada anteriormente pelo operador *new*
  - Devolve a memória ao sistema operacional

Alocação Dinâmica de Memória III

```
int main(){
    int i, n; double *nota = nullptr;
    cout << "Entre com o tamanho";
    cin >> n;
    // Aloca 8*n bytes de memoria
    nota = new double[n];
    for (i = 0; i < n; i++){
        cout << "Entre a nota: " << i+1 << endl;
        cin >> nota[i];
    }
    for (i = 0; i < n; i++)
        cout << nota[i] << " ";
    delete[] nota;
    return 0;
}
```

Alocação Dinâmica de Memória IV

	Comando	Liberação Memória
Alocação Estática de Memória	`int v[3];` Reserva 3 espaços de 4 bytes em **v**	O propio programa ao ser encerrado, se encarrega de liberar a memória alocada.
Alocação Dinâmica de Memória	`v = new int[n];` Reserva *n* espaços de 4 bytes em **v**	Se não for usar mais a variável *v*, então, é necessário empregar o comando `delete[]`.

Relação entre ponteiros, vetores e matrizes I

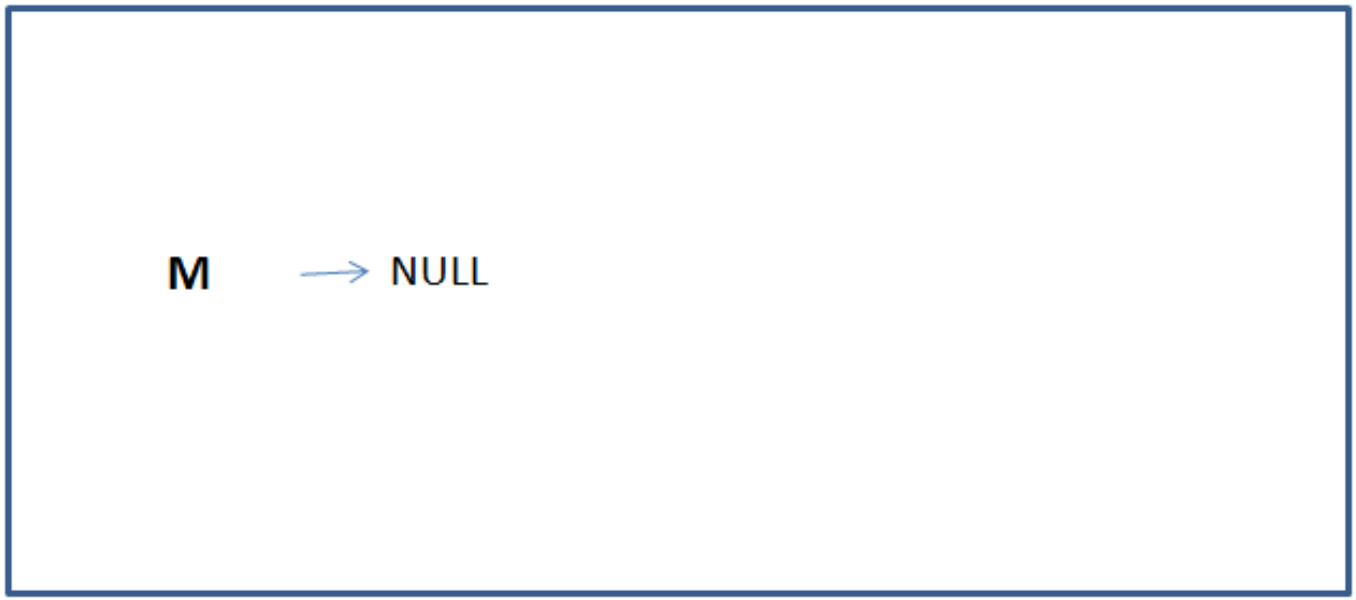
- Assim como é possível alocar memória em tempo de execução para armazenar um vetor, também, é possível construir uma matriz M com m linhas e n colunas. Os comandos para tal tarefa são como dados a seguir:

```
int main()
{
    int **M,
    int m, n, i;
    cout << "Entre com m e n";
    cin >> m >> n;
    // Vetor de endereços (os elementos são do tipo char*)
    M = new int*[m];
    // Cria para cada endereço um vetor de elementos int
    for (i = 0; i < m; i++)
        M[i] = new int[n]
    ...
    return 0;
}
```

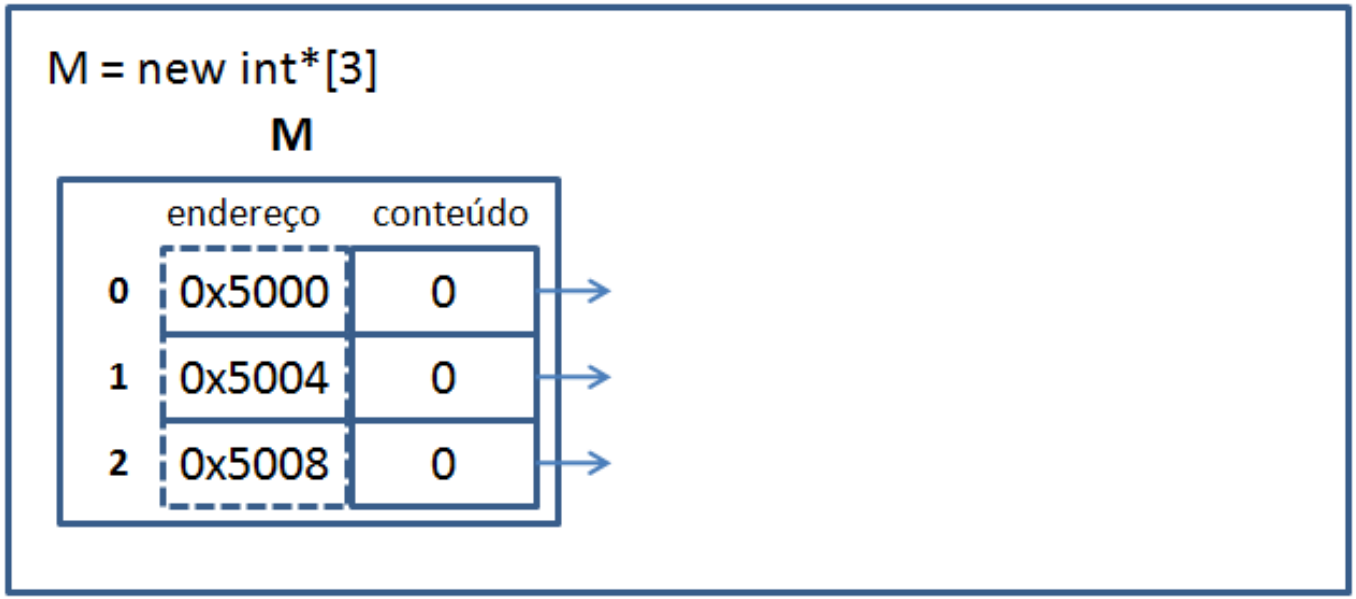
Relação entre ponteiros, vetores e matrizes II

- Vejamos um exemplo:

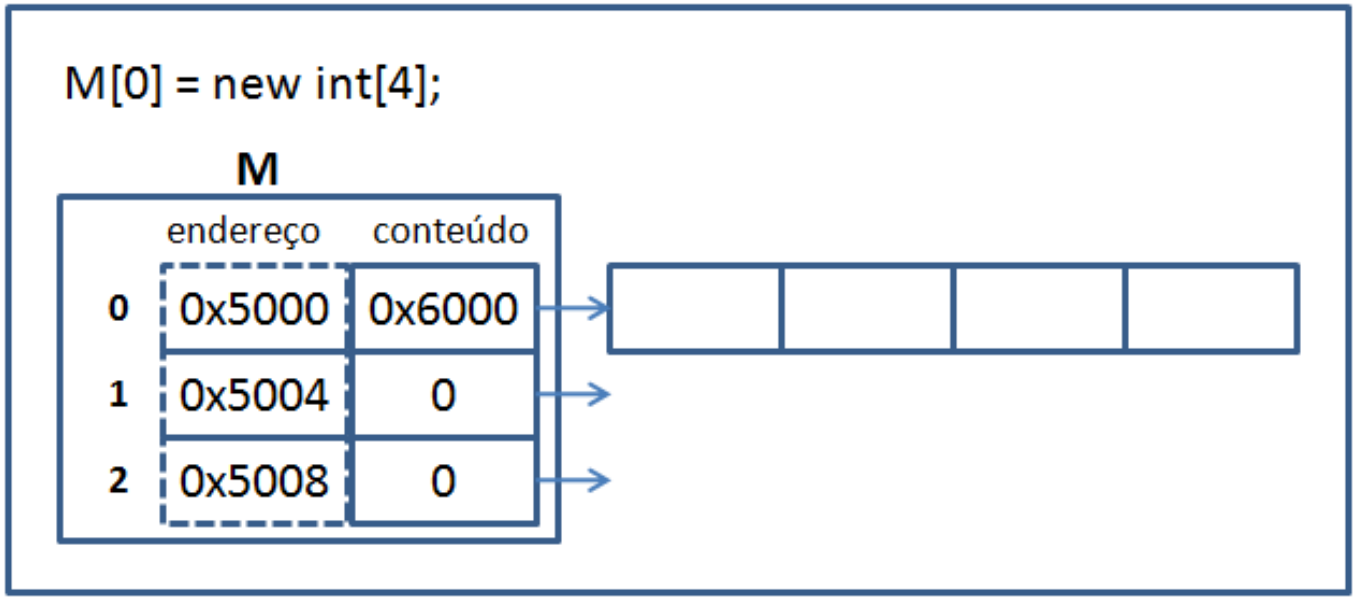
```
M = new int*[3];
```



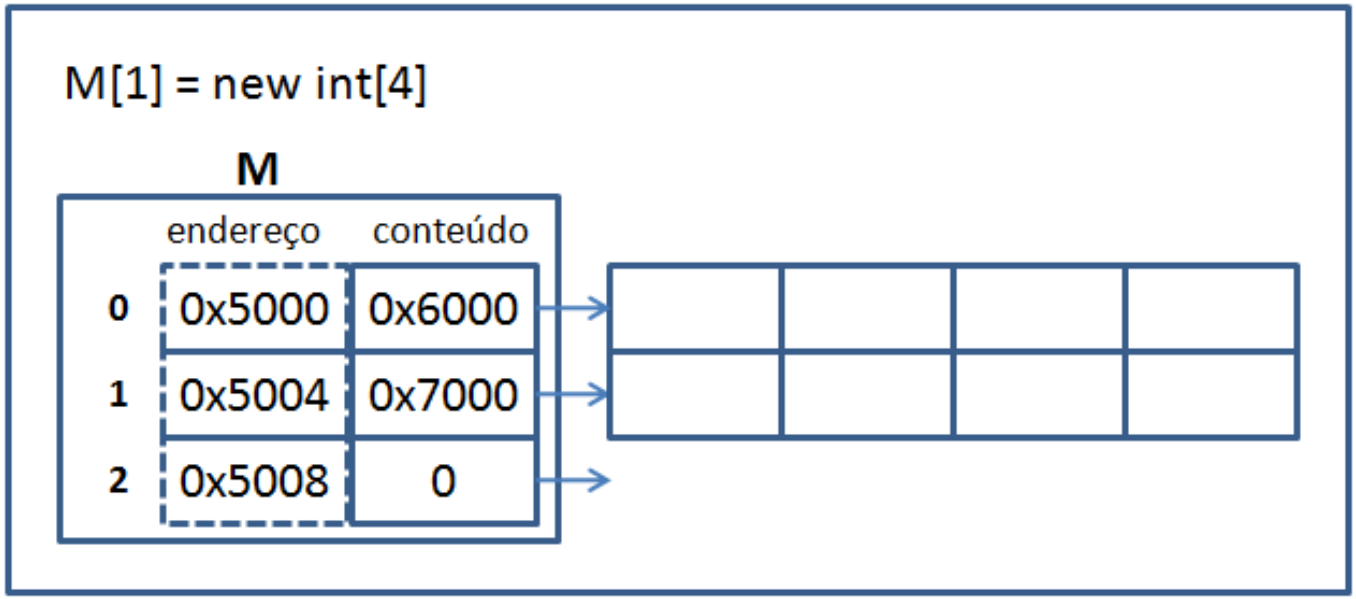
Relação entre ponteiros, vetores e matrizes III



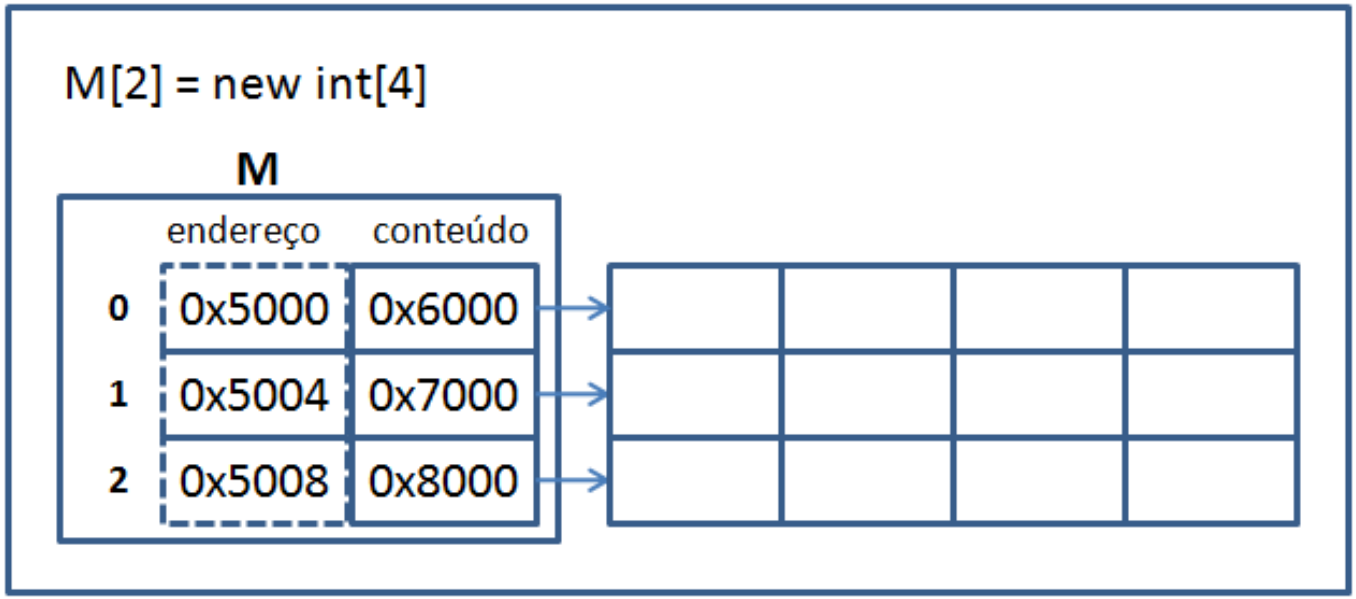
Relação entre ponteiros, vetores e matrizes IV



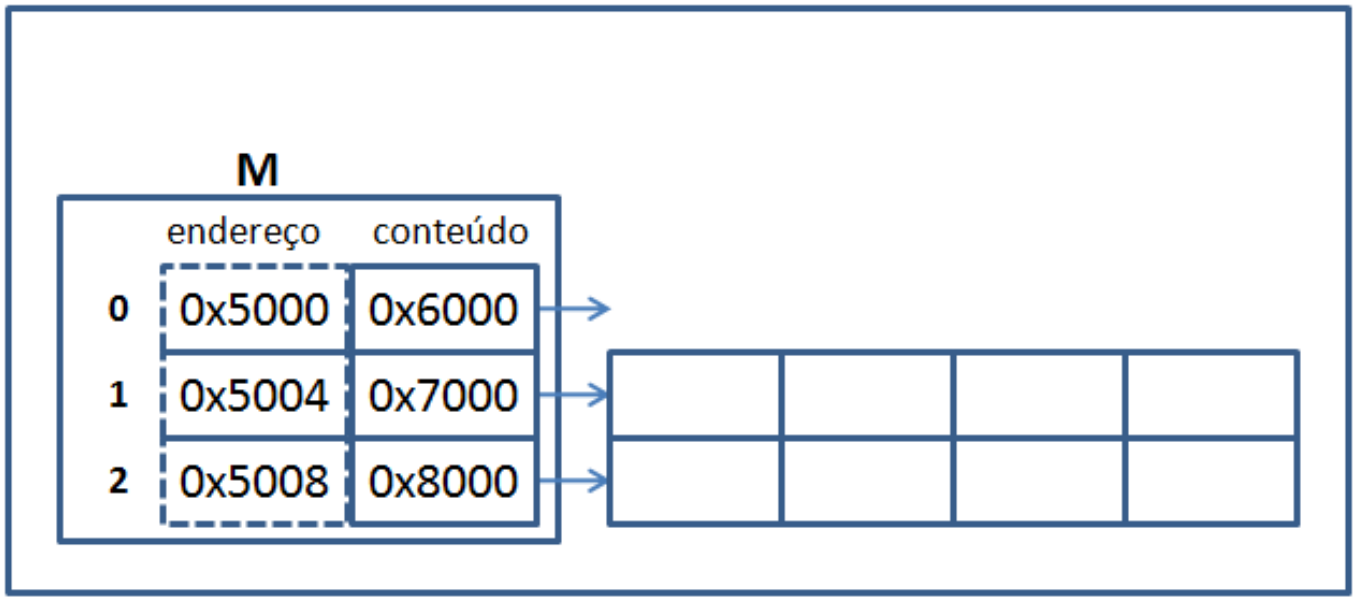
Relação entre ponteiros, vetores e matrizes V



Relação entre ponteiros, vetores e matrizes VI

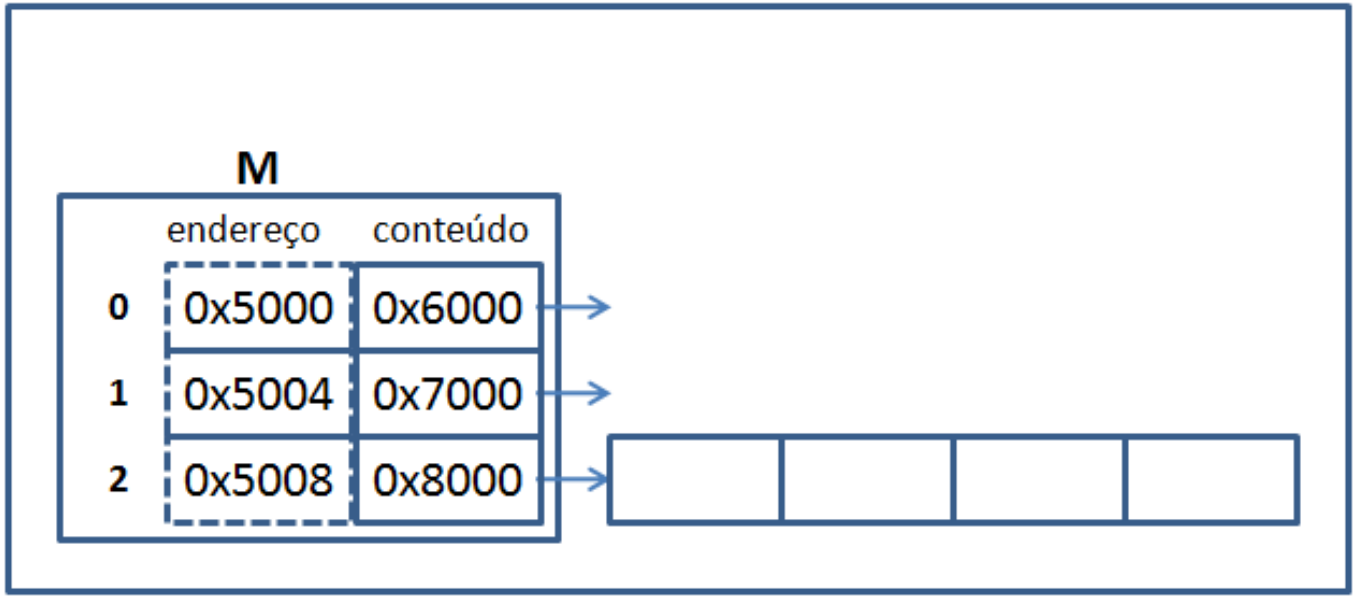


Liberação de memória I



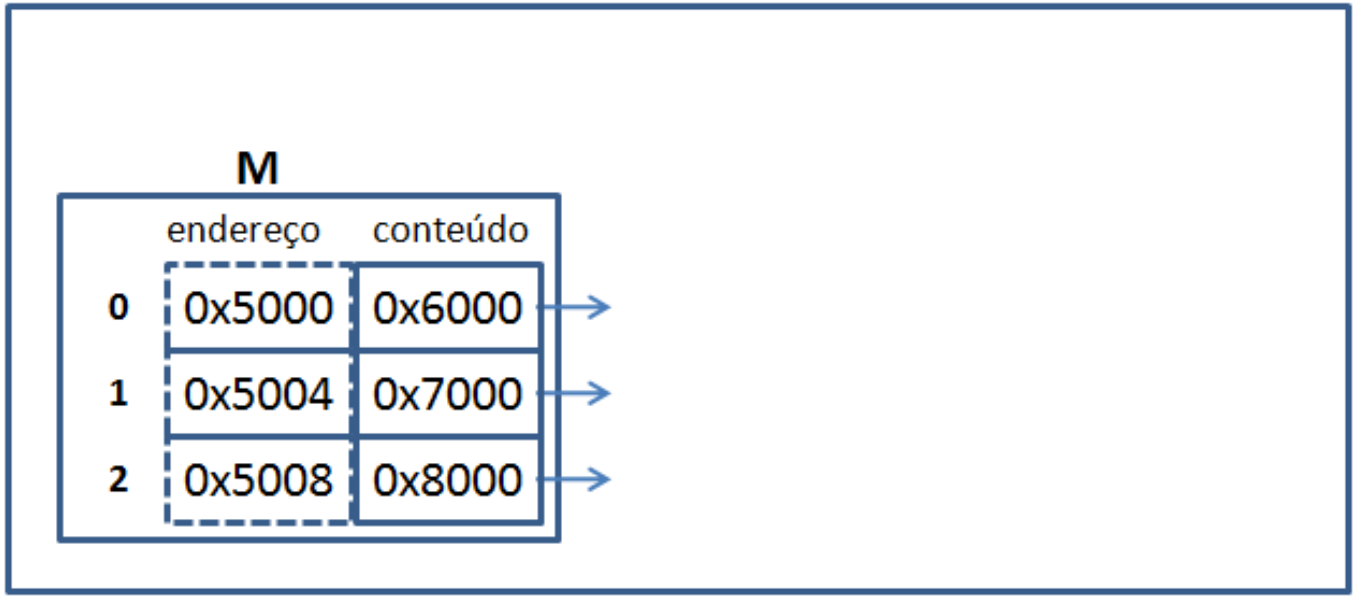
delete[] M[0]

Liberação de memória II



delete[] M[1]

Liberação de memória III



delete[] M[2]

Liberação de memória IV

delete[] M

Exercicios I

Inserir n notas de um total de m alunos

Exercicios II

```
int main()
{   int i, j, m, n; float **M = nullptr;
    cout << "Entre com m e n: ";
    cin >> m >> n;
    // Aloca m espaços tipo float *.
    M = new float*[m];
    // Aloca n espaços tipo float, cada M[i].
    for (i = 0; i < m; i++)
        M[i] = new float[n];
    // Preenchendo a matriz M usando índices: M[i][j].
    for (i=0; i < m; i++)
    {   cout << "Aluno: " << i+1 << endl;
        for (j=0; j < n; j++)
        {   cout << " Nota " << j+1;
            cin >> M[i][j];
        }
    }
    . . .
}
```

Exercicios III

```
    . . .
    // Impressão dos elementos de M, empregando ponteiros.
    for (i=0; i < m; i++)
    {   for (j=0; j < n; j++)
            cout << M[i][j] <<" ";
        cout << endl;
    }
    // Liberação de memória.
    // Liberando m vetores de tamanho n.
    if (M != nullptr){
        for (i=0; i < m; i++)
            if (M[i] != nullptr) delete[] M[i];
        // Liberando o vetor de ponteiros
        // de tamanho m.
        delete[] M;
    }
    return 0;
}
```

Exercicios IV

Criar a função que permite alocar uma matriz

```
#include <iostream>
using namespace std;

int** CriaMatriz(int, int);
void Apaga(int**);
int main(){
    int** Mat, lin, col;
    cout << "Digite a dimensao da matriz: ";
    cin >> lin >> col;
    Mat = CriaMatriz(lin, col);
    ...
    Apaga(Mat);
    return 0;
}
```

Exercicios V

```
int** CriaMatriz(int m, int n)
{
    int** M = nullptr;
    M = new int*[m];
    M[0] = new int[m*n];
    for (int i = 1; i < m; i++)
    {
        M[i] = &M[0][i*n];
    }
    return M;
}
```

Exercicios VI

```
void Apaga(int** M)
{
    if (M != nullptr)
    {
        if (M[0] != nullptr) delete[] M[0];
        delete[] M;
    }
}
```

Sobrecarga de funções I

- Sobrecarregar funções significa criar uma familia de funções que tenham o mesmo nome, mas uma lista de parâmetros diferente
- Por exemplo, vamos considerar uma função que calcula o cubo de um número
- Que tipo de número? Inteiro ou real?
- Criamos uma função para cada, e o próprio programa escolhe a mais adequada baseada no tipo do parâmetro.

Sobrecarga de funções II

```
#include <iostream>
using namespace std;

int cubo(int num){
    return num * num * num;
}
int cubo(float num){
    return num * num * num;
}
int cubo(int double){
    return num * num * num;
}

int main(){
    float num = 56.7;
    cout << cubo(5) << endl
    cout << cubo(num) << endl
    cout << cubo(458.65) << endl
    return 0;
}
```

Estruturas I

- Saber lidar com estruturas é meio caminho para aprender a lidar com classes e objetos
- Definir uma estrutura define um molde para criação de variáveis
- A linguagem C++ expande as capacidade das estruturas
  - Em C as estruturas somente armazenam dados
  - Em C++ as estruturas podem armazenar dados e funções

Estruturas II

- A diferença na sintaxe de estruturas e classes é mínima
  - Basicamente, classes definem membros privados por padrão;
  - Estruturas definem membros públicos por padrão
- A maior parte dos programadores em C++ utiliza estruturas para dados e classes para dados e funções

Estruturas III

- Problemas reais
  - Temos coleções de dados que são de **tipos diferentes**
  - Exemplo: ficha de um cadastro de cliente
    - Nome: *string*
    - Endereço: *string*
    - Telefone: *string*
    - Salário: *float*
    - Idade: *int*

Estruturas IV

- Registro (ou *struct*)
  - Tipo de dado estruturado heterogêneo
    - Coleção de variáveis referenciadas sobre um mesmo nome
  - Permite agrupar dados de diferentes tipos numa mesma estrutura (ao contrário de matrizes que possuem elementos de um mesmo tipo)
    - Cada componente de um registro pode ser de um tipo diferente
    - Estes componentes são referenciados por um nome

Estruturas V

- Os elementos do registro
  - São chamados de campos ou membros da *struct*
- É utilizado para armazenar informações de um mesmo objeto
- Exemplos:
  - carro $\rightarrow$ cor, marca, ano, placa
  - pessoa $\rightarrow$ nome, idade, endereço

Estruturas VI

- Campo (*Field*)
  - Conjunto de caracteres com o mesmo significado
  - Exemplo: nome
- Registro (*struct* ou *record*)
  - Conjunto de campos relacionados
  - Exemplo: nome, endereço, telefone, salários e idade de uma pessoa

Sintaxe na Linguagem C/C++ I

- A palavra reservada *struct* indica ao compilador que está sendo criada uma estrutura

- Uma estrutura deve ser declarada após incluir as bibliotecas e antes do *main*

```
struct <identicador_struct>
{
    tipo <nome_variável_campo1>;
    tipo <nome_variável_campo2>;
    ...
}<variáveis_estrutura>;

struct <identificador_struct> <var1>, <var2>;
```

Sintaxe na Linguagem C/C++ II

- Se o compilador C for compatível com o padrão C ANSI
  - Informação contida em uma *struct* pode ser atribuída a outra *struct* do mesmo tipo
  - Não é necessário atribuir os valores de todos os elementos/campos separadamente
  - Por exemplo: `<var1> = <var2>;`
    - Todos os campos de `<var1>` receberão os valores correspondentes dos campos de `<var2>`

Sintaxe na Linguagem C/C++ III

- Para acessar os campos da *struct*
  - Utiliza-se o nome da variável *struct*, seguido de ponto, seguido do nome do campo
  - Por exemplo: `<var1>.<nome_campo>`

Sintaxe na Linguagem C/C++ IV

- Por exemplo um *struct* endereço que guarda os elementos nome, rua, cidade, estado e cep

```
struct endereco
{
    string nome;
    string rua;
    ...
    long int cep;
};
```

  - Foi feita apenas a declaração da *struct*, ainda não foi criada nenhuma variável da *struct* endereço
  - o comando para declarar uma variável com esta *struct* é: `struct endereco info_end;`

Sintaxe na Linguagem C/C++ V

- Já vimos que para acessar os membros de uma *struct* deve-se usar `nome_variável.nome_membro`
- Portanto, considerando o exemplo anterior
  - Para inicializar o cep da variável `info_end` que é uma variável da *struct* endereço se faria:
    `info_end.cep = 123456;`
  - Para obter o nome da pessoa e colocar na *string* nome da *struct* se poderia utilizar:
    `getline(cin, info_end.nome);`
  - Para imprimir o endereço seria:
    `cout << info_end.rua;`

Ponteiros: Passagem por valor e por referência I

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 II

```
typedef struct Pessoa
{
    string nome;
    double salario;
    char sexo;
}PE;

void Insere(PE*, int);
void Print(PE*, int);
int main()
{
    int n;
    PE *trab = nullptr;
    cout << "Quantidade de pessoas";
    cin >> n;
    trab = new PE[n];
    Insere(trab, n);
    Print(trab, n);
    if (trab != nullptr) delete[] trab;
    return 0;
}
```

Ponteiros: Passagem por valor e por referência III

```
void Insere(PE* vet, int n){
    int i, valores;
    for (i = 0; i < n; i++) {
        cout << "Cadastro numero " << i+1 << endl;
        cout << "Insere nome: ";
        getline(cin, vet[i].nome);
        do{
            cout << "Insere salario: ";
            cin >> vet[i].salario;
            if ( cin.fail() ){
                cout << "Nao eh um numero \n";
                cin.clear(); cin.ignore(100, '\n');
            }
            else break;
        }while(true);
        cout << "Insere sexo: "; cin >> vet[i].sexo;
        cin.ignore(numeric_limits
           <std::streamsize>::max(), '\n');
    }
}
```

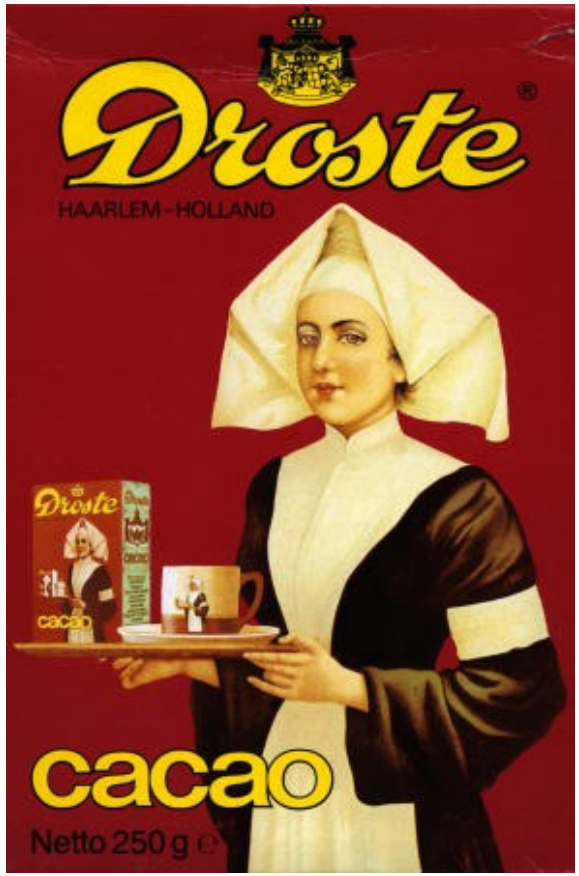
Ponteiros: Passagem por valor e por referência IV

```
void Print(PE* vet, int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        cout << vet[i].nome << vet[i].salario
             << vet[i].sexo << endl;
    }
}
```

Recursividade I

- Recursividade: é um método de programação no qual uma função pode chamar a si mesma
- Muitas estruturas têm natureza recursiva:
  - Estruturas encadeadas
  - Fatorial, serie Fibonacci
  - Uma pasta que contém outras pastas e arquivos

Recursividade II



Uma forma visual de recursividade conhecida como efeito Droste

Recursividade III

Recursividade IV

- Recursão matemática
  - Como definir recursivamente a seguinte soma?

$$\sum_{k=m}^{n} k = m + (m+1) + \ldots + (n-1) + n$$

Recursividade V

- Primeira definição recursiva

$$\sum_{k=m}^{n} k \begin{cases} m & \text{se } n = m \\ n + \sum_{k=m}^{n-1} k & \text{se } n > m \end{cases}$$

Recursividade VI

- Recursão na computação

```
float soma(float m, float n);
int main(){
    float s;
    s = soma(5, 8);
    cout << s;
}
float soma(float m, float n)
{
    if (m == n)
        return m;
    else
        return (n + soma(m, n-1));
}
```

Na tela é mostrado:
26

Pilha de execução I

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.

```
S = soma(5, 8);
```

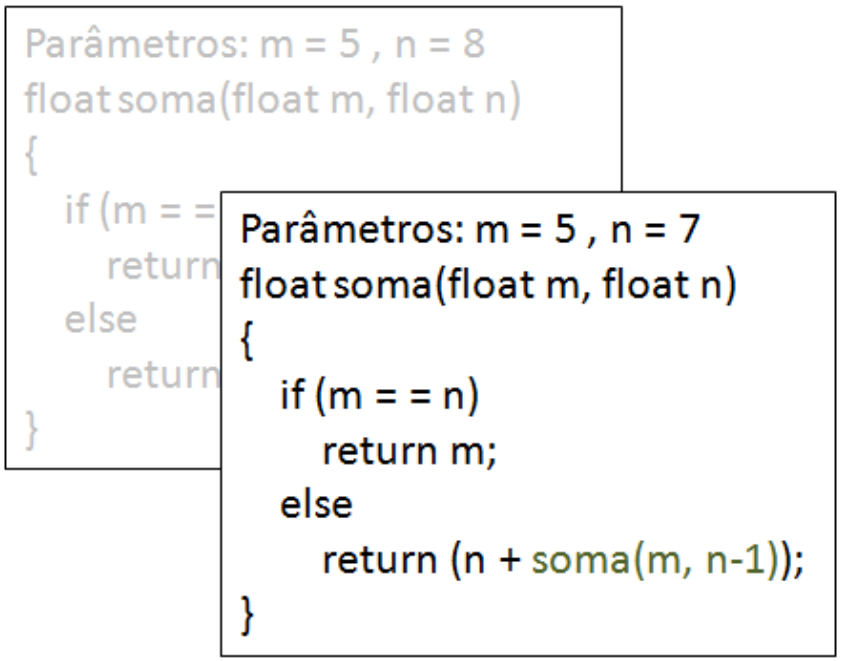
Pilha de execução II

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.

```
Parâmetros: m = 5 , n = 8
float soma(float m, float n)
{
   if (m = = n)
      return m;
   else
      return (n + soma(m, n-1));
}
```

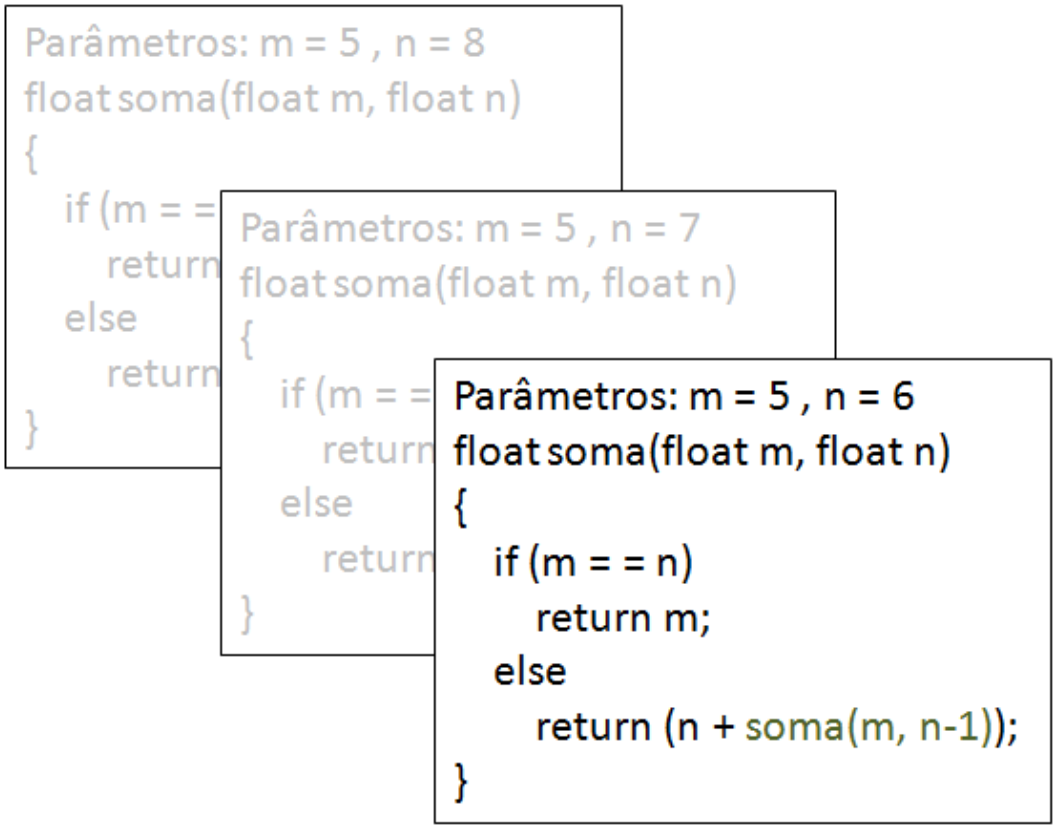
Pilha de execução III

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



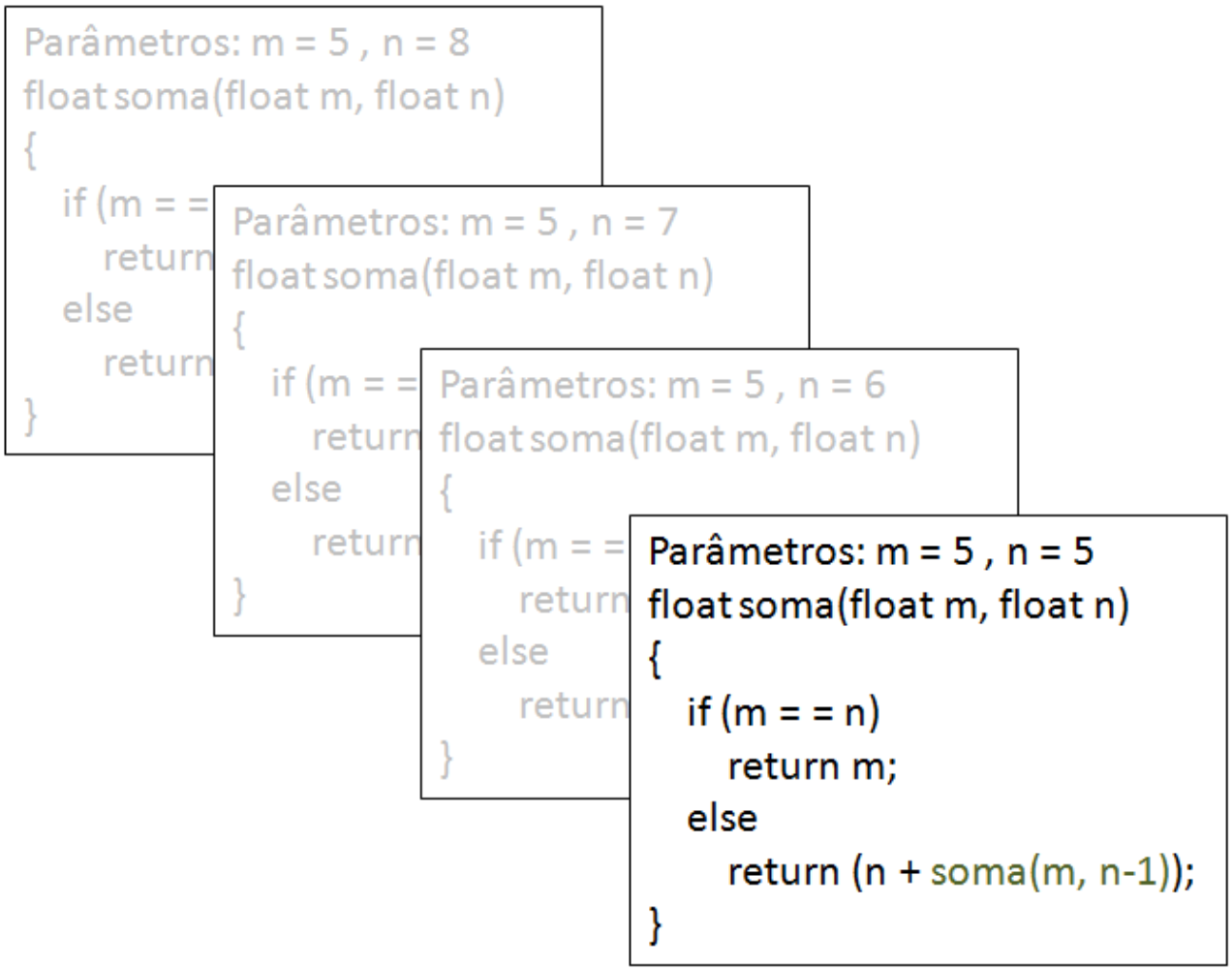
Pilha de execução IV

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



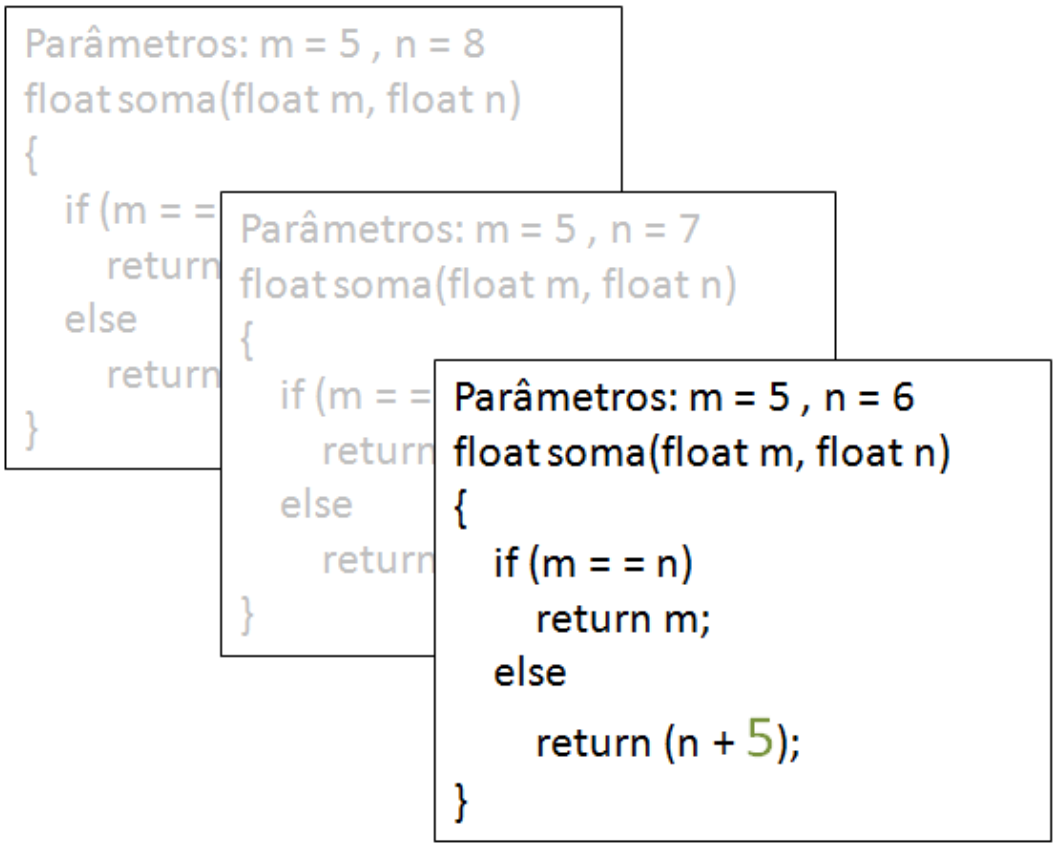
Pilha de execução V

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



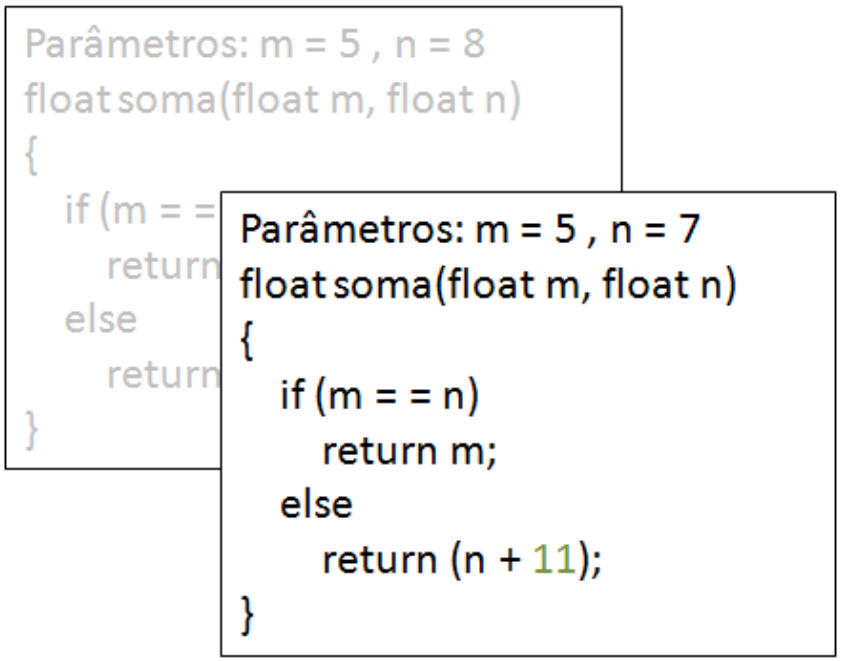
Pilha de execução VI

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



Pilha de execução VII

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.



Pilha de execução VIII

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.

```
Parâmetros: m = 5 , n = 8
float soma(float m, float n)
{
   if (m = = n)
      return m;
   else
      return (n +18);
}
```

Pilha de execução IX

A cada chamada de função o sistema reserva espaço para **parâmetros**, **variáveis locais** e **valor de retorno**.

S = 26

Estouro de pilha de execução I

- O que acontece se a função não tiver um caso base (ponto de parada)?
- O sistema de execução não consegue implementar infinitas chamadas

Fatorial I

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n * (n-1)! & \text{se } n > 0 \end{cases}$$

Fatorial II

```
int fatorial(int n){
    if (n == 0)
        return 1;
    else
        return n * fatorial(n-1);
}
```

Potência I

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ x * x^{n-1} & \text{se } n > 0 \end{cases}$$

Potência II

```
double potencia(double x, int n)
{
    if (n == 0)
        return 1;
    else
        return x * potencia(x, n-1);
}
```

Exemplos I

O máximo divisor comum de dois números inteiros positivos pode ser calculado, utilizando o método de Euclides, cujo algoritmo é dado pela seguinte relação de recorrência:

$$mcd(m,n) = \begin{cases} m, & \text{se } n = 0 \\ mcd(n, m\%n), & \text{se } n \neq 0 \end{cases}$$

Exemplos II

```
int mdc(int m, int n)
{
    if (n == 0)
        return m;
    else
        return mdc(n, m % n);
}
```

Exemplos III

Escrever um programa que leia do teclado dois números inteiros positivos que correspondem ao numerador e denominador de uma fracção, permita reduzir a fracção e escreva no monitor a fracção reduzida assim como o seu quociente.

Exemplos IV

```
int mdc(int , int );
void reduzir(int , int , int&, int&);
int main()
{
    int num,den, n_num, n_den;
    cout << "Inserir fraccao: numerador, denominador \n";
    cin >> num >> den;
    reduzir(num, den, n_num, n_den);
    cout << num << "/" << den << "="
         << n_num << "/" << n_den;
    return 0;
}
```

Exemplos V

```
void reduzir(int num, int den, int& n_num, int& n_den)
{
    int fator = mdc (num, den);
    n_num = n_num / fator;
    n_den = n_den / fator;
}
int mdc(int m, int n)
{
    if (n == 0)
        return m;
    else
        return mdc(n, m  % n);
}
```

Exemplos VI

Encontrar a soma dos elementos de um vetor

Exemplos VII

```
int soma(int vet[], int n);
int main()
{
    int A[5] = {3, 5, 8, 23, 9};
    cout << soma(A, 5);
    return 0;
}
int soma(int vet[], int n)
{
    if (n == 1)
        return vet[0];
    else
        return vet[n-1] + soma(vet, n-1);
}
```

FIM