

INF 112 - Programação II

Alocação Dinâmica e Armazenamento de Dados em Memória

Introdução

- Sempre que escrevemos um programa, é preciso reservar espaço para as informações que serão processadas
- Para isso utilizamos as variáveis
 - Uma variável é uma posição de memória que armazena uma informação que pode ser modificada pelo programa
 - Ela deve ser definida antes de ser usada

Introdução

- A linguagem de programação C/C++ permite dois tipos de alocação de memória: estática e dinâmica
- Na **alocação estática**, o espaço de memória para as variáveis é reservado ainda na compilação, não podendo ser alterado depois.

```
int a;  
int b[10];
```

Alocação Dinâmica

- A **Alocação Dinâmica** permite ao programador criar “variáveis” em tempo de execução, ou seja, alocar memória para novas variáveis quando o programa está sendo executado, e não apenas quando se está escrevendo o programa.

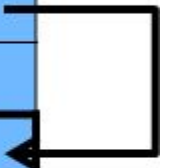
Alocando Memória

Memória		
#	var	conteúdo
119		
120		
121	int *n	NULL
122		
123		
124		
125		
126		
127		
128		
129		

Alocando 5
posições de
memória em int * n



Memória		
#	var	conteúdo
119		
120		
121	int *n	#123
122		
123	n[0]	11
124	n[1]	25
125	n[2]	32
126	n[3]	44
127	n[4]	52
128		
129		



Alocação Dinâmica

- Espaço de memória é requisitado em tempo de execução e permanece reservado até que seja explicitamente liberado

Alocação Dinâmica em C

- A linguagem C ANSI usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na `stdlib.h`:
 - `malloc`
 - `calloc`
 - `realloc`
 - `free`

Alocação Dinâmica em C

- Exemplo: alocar um vetor de inteiros de 10 elementos.
- malloc retorna o endereço da área alocada para armazenar os valores inteiros
- o ponteiro de inteiro recebe o endereço inicial do espaço alocado

```
int *a;  
v = (int *) malloc (10 * sizeof(int));
```


Alocação Dinâmica em C

- Ao contrário da memória alocada estaticamente, a memória alocada dinamicamente precisa ser “liberada”

```
free (v);
```

Alocação Dinâmica em C++

- O operador **new** retorna o endereço onde começa o bloco de memória que atribuímos em um ponteiro.
- A função deste operador é alocar memória dinamicamente.

```
int *valor = new int;  
char *letra = new char('J');  
int *vetor = new int[10];
```


Alocação Dinâmica em C++

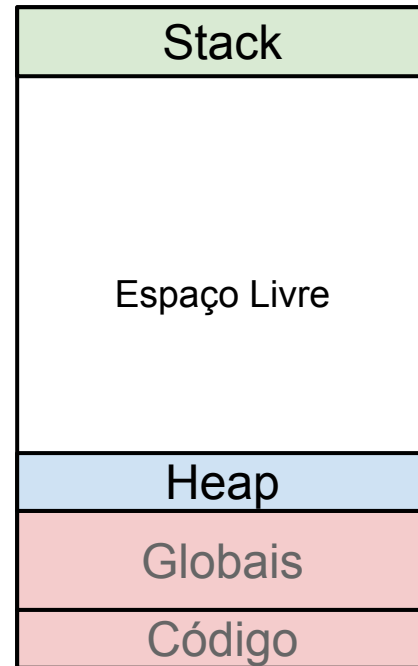
- É necessário liberar a memória alocada dinamicamente por meio do operador **delete**.

```
delete valor;  
delete []vetor;
```

- Por que é importante liberar a memória?
 - memory leak.

Um programa na memória

- Podemos representar um programa em regiões como:
 - Pilha (Stack)
 - Heap
 - Código
 - Globais
- 
- O diagrama ilustra a organização da memória em regiões distintas. É composto por uma barra vertical dividida em quatro seções. A seção superior é uma faixa verde rotulada 'Stack'. Abaixo dela é uma grande área branca rotulada 'Espaço Livre'. Segue-se uma faixa azul rotulada 'Heap'. A base da barra é uma faixa vermelha, que não possui rótulo, mas corresponde à categoria 'Globais' mencionada no texto adjacente.



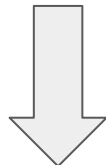
Compilador

```
$ g++ hello.cpp -o hello
```

Programa

Compilador

```
$ g++ hello.cpp -o hello
```

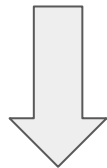


```
00000000000008a5 <main>:
 8a5: 55                push   %rbp
 8a6: 48 89 e5          mov     %rsp,%rbp
 8a9: 48 8d 35 15 01 00 00 lea     0x115(%rip),%rsi    # 9c5
<_ZStL19piecewise_construct+0x1>
 8b0: 48 8d 3d a9 07 20 00 lea     0x2007a9(%rip),%rdi    # 201060
<_ZSt4cout@@GLIBCXX_3.4>
 8b7: e8 d4 fe ff ff    callq  790
<_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
 8bc: 48 89 c2          mov     %rax,%rdx
 8bf: 48 8b 05 0a 07 20 00 mov     0x20070a(%rip),%rax    # 200fd0
<_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GLIBCXX_3.4>
 8c6: 48 89 c6          mov     %rax,%rsi
 8c9: 48 89 d7          mov     %rdx,%rdi
 8cc: e8 cf fe ff ff    callq  7a0 <_ZNSolsEPFRSoS_E@plt>
 8d1: b8 00 00 00 00    mov     $0x0,%eax
 8d6: 5d                pop     %rbp
 8d7: c3                retq
```

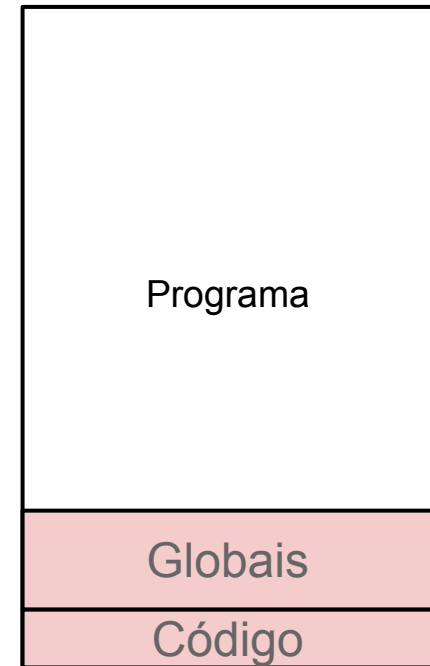
Programa

Compilador

```
$ g++ hello.cpp -o hello
```



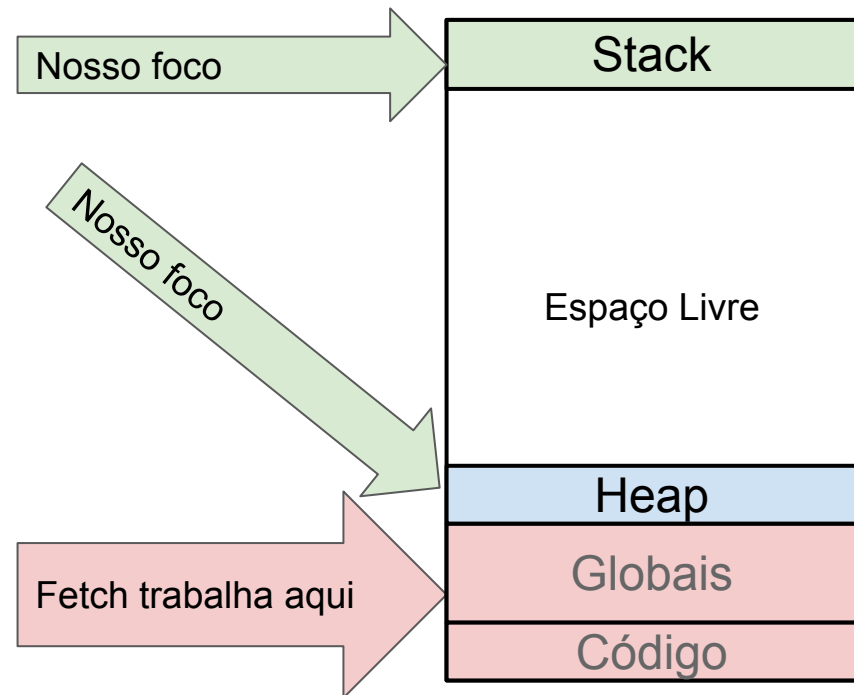
```
00000000000008a5 <main>:
 8a5: 55                push   %rbp
 8a6: 48 89 e5          mov    %rsp,%rbp
 8a9: 48 8d 35 15 01 00 00 lea    0x115(%rip),%rsi    # 9c5
<_ZStL19piecewise_construct+0x1>
 8b0: 48 8d 3d a9 07 20 00 lea    0x2007a9(%rip),%rdi    # 201060
<_ZSt4cout@@@GLIBCXX_3.4>
 8b7: e8 d4 fe ff ff    callq  790
<_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
 8bc: 48 89 c2          mov    %rax,%rdx
 8bf: 48 8b 05 0a 07 20 00 mov    0x20070a(%rip),%rax    # 200fd0
<_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_0_ES6_@GLIBCXX_3.4>
 8c6: 48 89 c6          mov    %rax,%rsi
 8c9: 48 89 d7          mov    %rdx,%rdi
 8cc: e8 cf fe ff ff    callq  7a0 <_ZNSolsEPFRSoS_E@plt>
 8d1: b8 00 00 00 00    mov    $0x0,%eax
 8d6: 5d                pop    %rbp
 8d7: c3                retq
```



Voltando para a memória

- Vamos focar no Stack/Heap

- Onde moram “os dados”



- A parte de código não muda muito de uma linguagem para outra (compilador)

Compilando

```
$ g++ programa.cpp -o programa
```

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    int x = 7;
    int y;
    y = function(x);
    x++;
    y--;
    return 0;
}
```

nome	ender.	valor
	0x0048	
	0x0044	
	0x0040	
	0x003c	
	0x0038	
	0x0034	
	0x0030	
	0x002c	
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

Compilando

```
$ g++ programa.cpp -o programa
```

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    int x = 7;
    int y;
    y = function(x);
    x++;
    y--;
    return 0;
}
```



nome	ender.	valor
	0x0048	
	0x0044	
	0x0040	
	0x003c	
	0x0038	
	0x0034	
	0x0030	
	0x002c	
	0x0028	
	0x0024	
	0x0020	
	0x001c	
	0x0018	
	0x0014	
	0x0010	
	0x000c	
	0x0008	
	0x0004	
	0x0000	

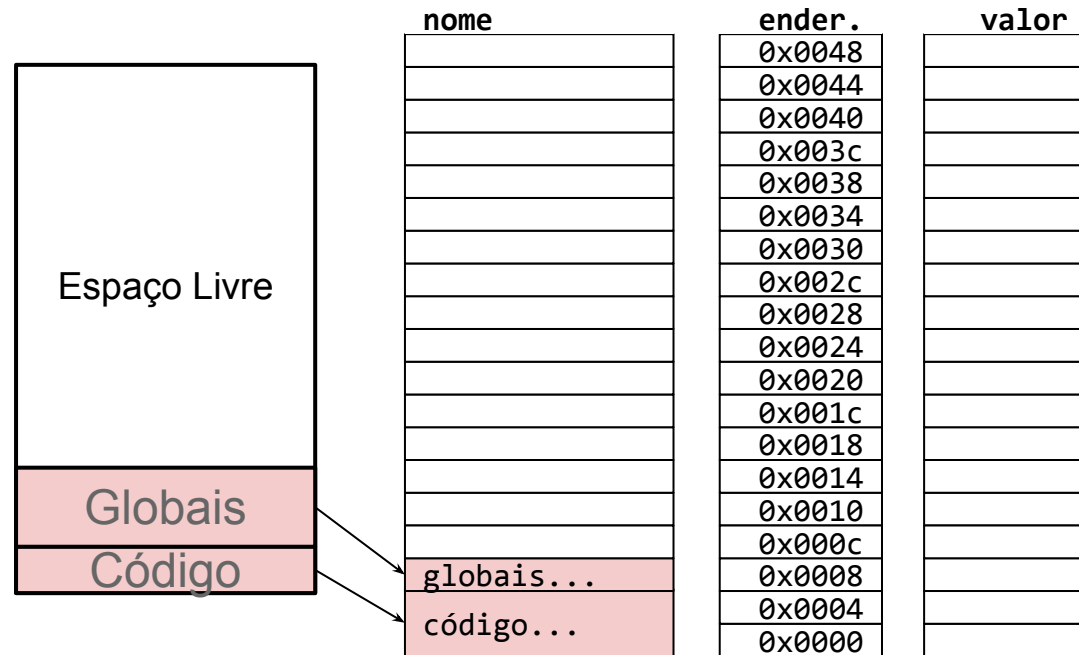
Dando início

```
$ ./programa
```

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    int x = 7;
    int y;
    y = function(x);
    x++;
    y--;
    return 0;
}
```



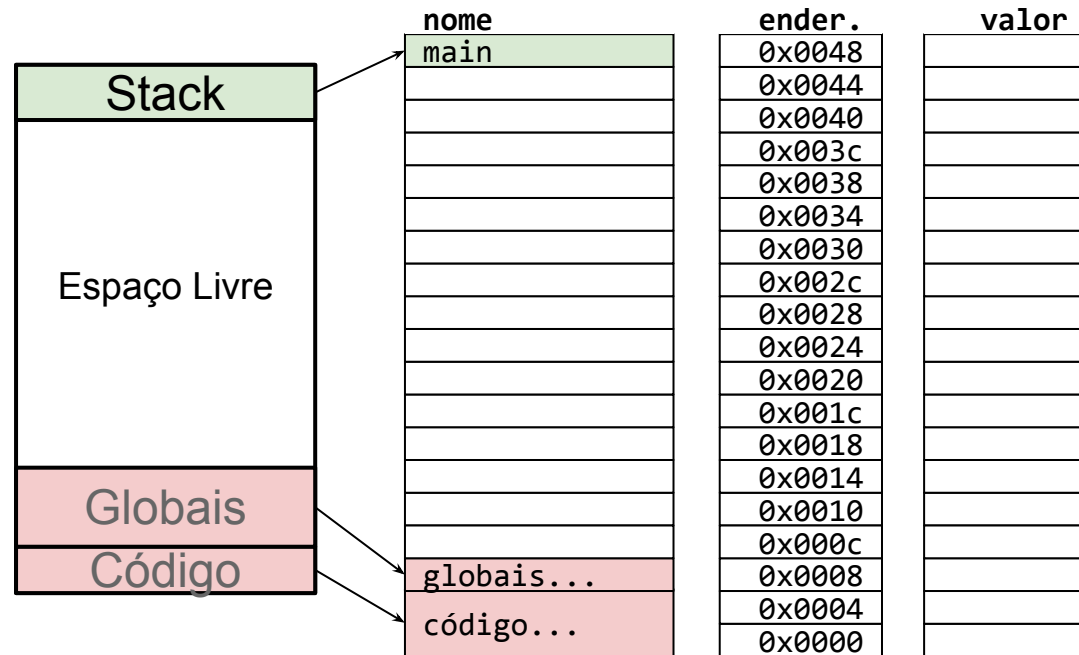
Stack

Colocamos o main na pilha

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

→ int main() {
    int x = 7;
    int y;
    y = function(x);
    x++;
    y--;
    return 0;
}
```



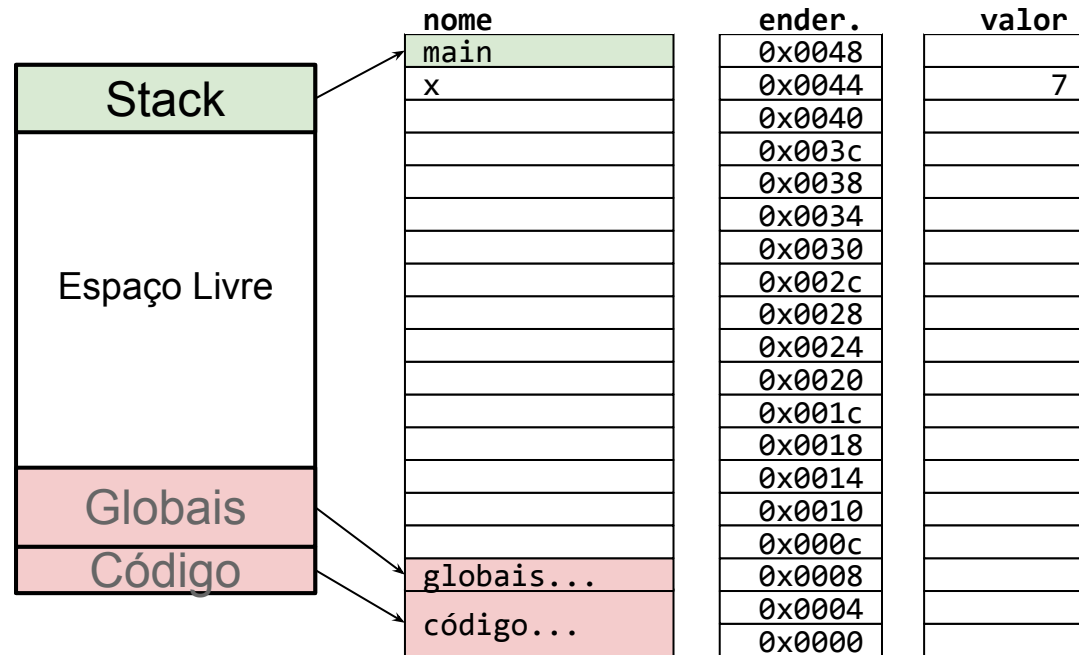
Stack

x = 7

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    → int x = 7;
    int y;
    y = function(x);
    x++;
    y--;
    return 0;
}
```



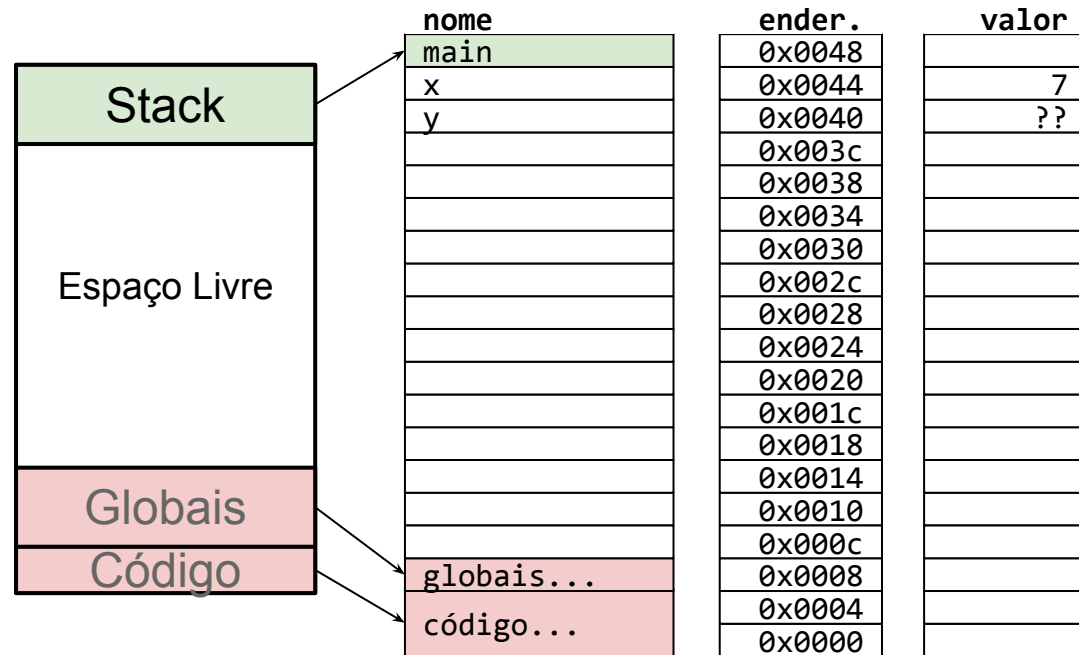
Stack

y = lixo

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    int x = 7;
    → int y;
    y = function(x);
    x++;
    y--;
    return 0;
}
```



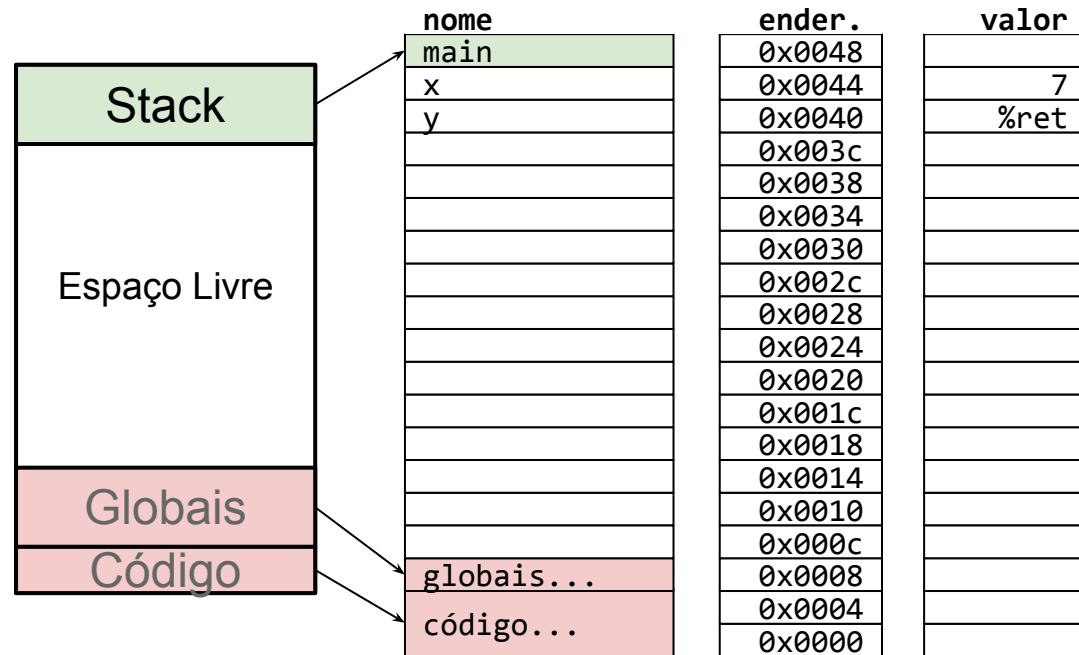
Stack

y = retorno de function. Representado por %ret

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    int x = 7;
    int y;
    → y = function(x);
    x++;
    y--;
    return 0;
}
```



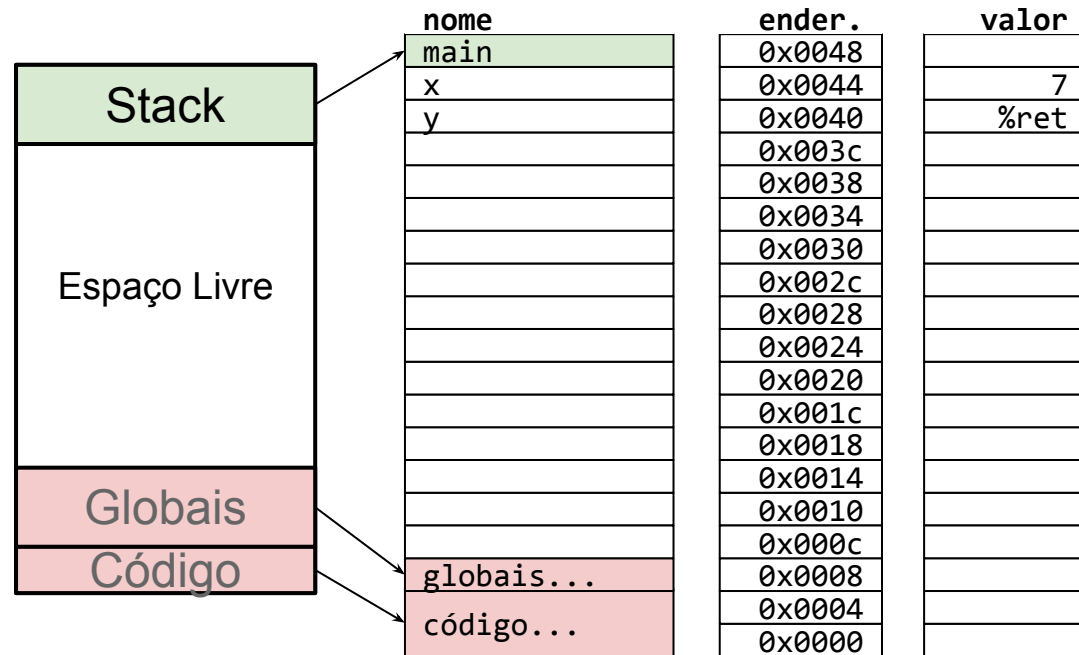
Stack

y = retorno de function. Representado por %ret

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    int x = 7;
    int y;
    → y = function(x);
    x++;
    y--;
    return 0;
}
```



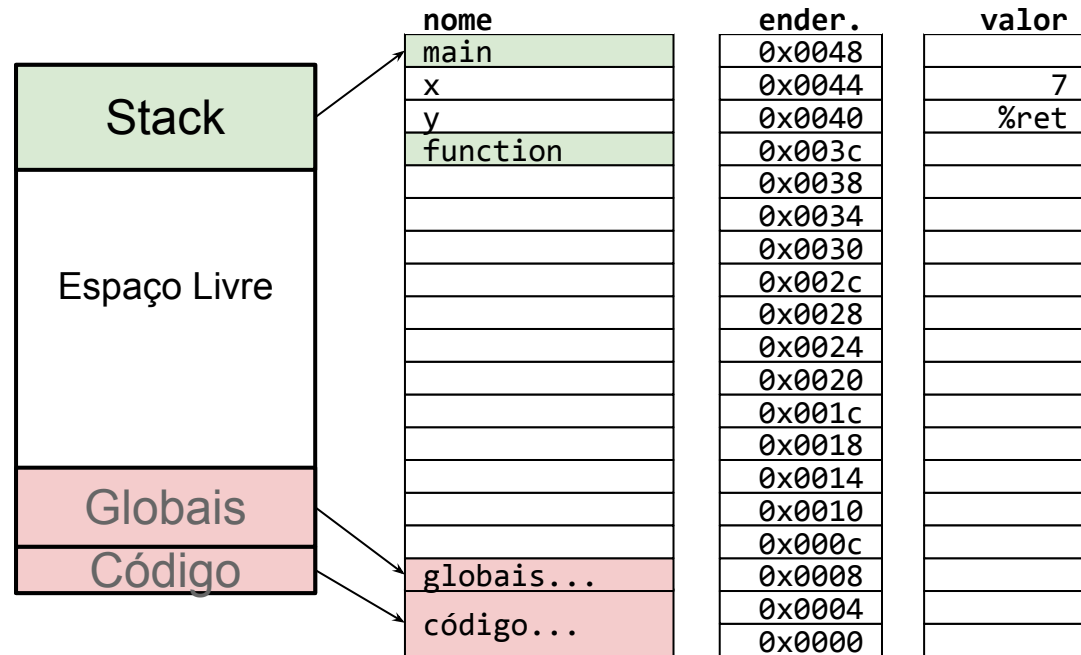
Stack

chamamos a função function

```
#include <iostream>
```

```
➔ int function(int &f) {  
    f=f+3;  
    return f;  
}
```

```
int main() {  
    int x = 7;  
    int y;  
    y = function(x);  
    x++;  
    y--;  
    return 0;  
}
```



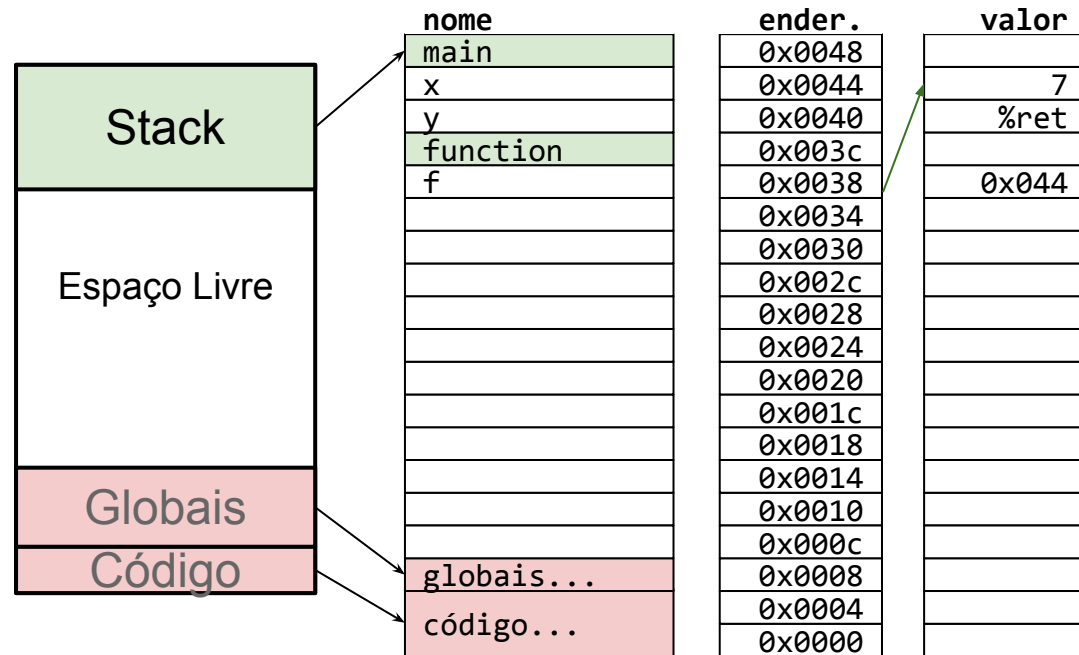
Stack

empilhamos f. & é uma referência

```
#include <iostream>
```

```
➔ int function(int &f) {  
    f=f+3;  
    return f;  
}
```

```
int main() {  
    int x = 7;  
    int y;  
    y = function(x);  
    x++;  
    y--;  
    return 0;  
}
```



Stack

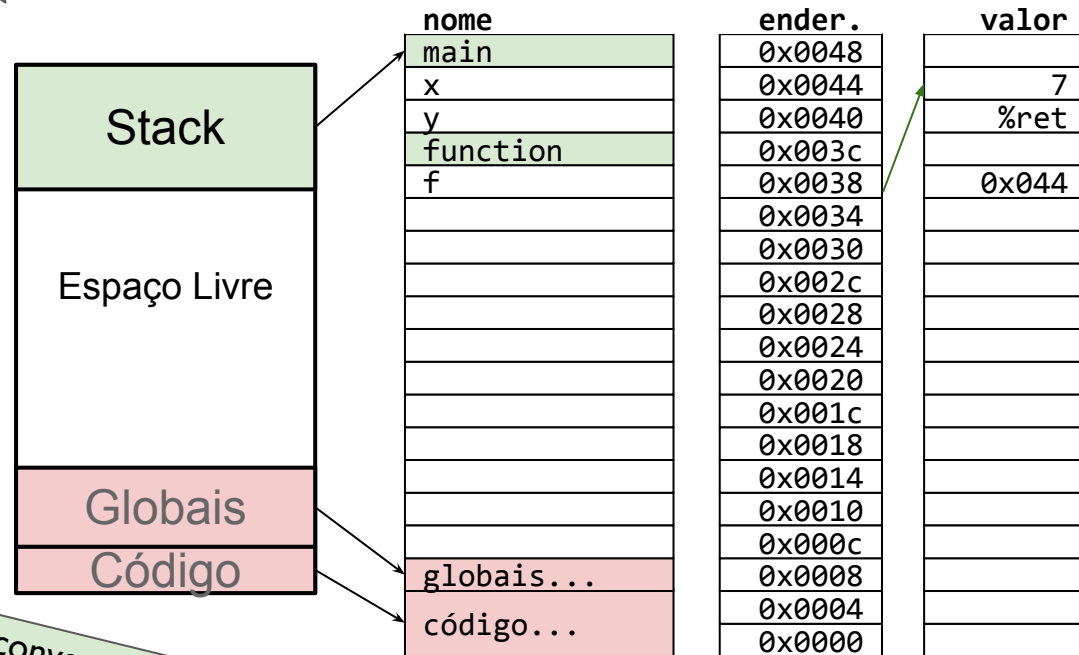
empilhamos f. & é uma referência

```
#include <iostream>
```

```
int function(int &f) {  
    f=f+3;  
    return f;  
}
```

```
int main() {  
    int x = 7;  
    int y;  
    y = function(x);  
    x++;  
    y--;  
    return 0;  
}
```

Usando int& convertemos para uma referência diretamente



Usando int& convertemos para uma referência diretamente

Stack

diferente do ponteiro em C, referências são imutáveis

```
#include <iostream>
```

```
int function(int &f) {
```

```
    f=f+3;
```

```
    return f;
```

```
}
```

```
int main() {
```

```
    int x = 7;
```

```
    int y;
```

```
    y = function(x);
```

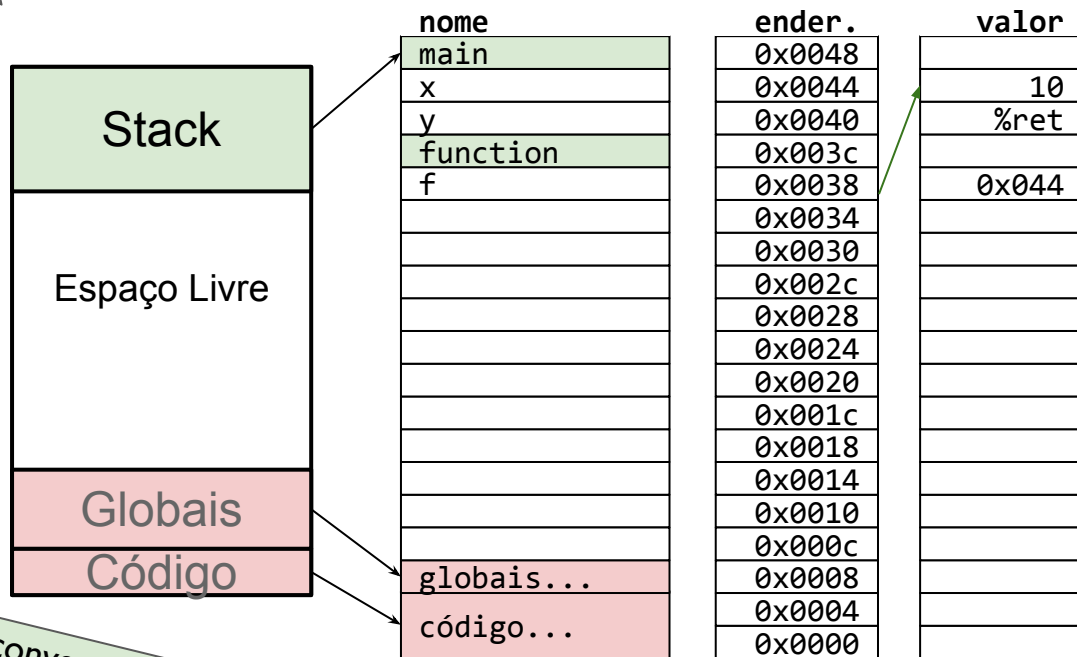
```
    x++;
```

```
    y--;
```

```
    return 0;
```

```
}
```

Usando int& convertemos para uma referência diretamente



Usando int& convertemos para uma referência diretamente

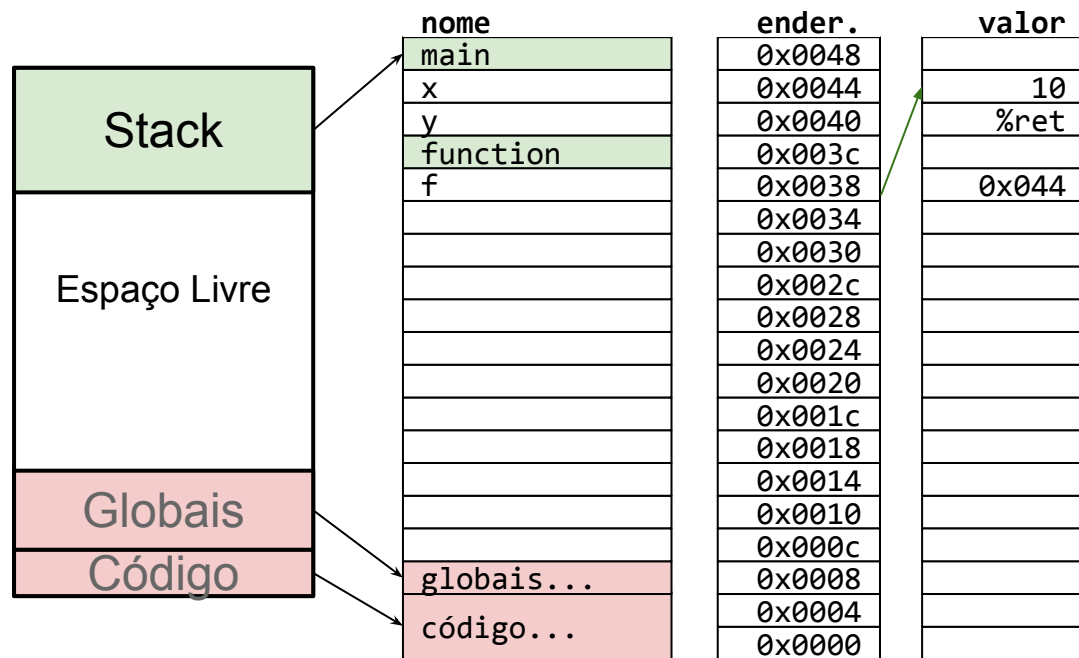
Stack

Uma vantagem em C++, fim do uso excessivo de *

```
#include <iostream>
```

```
int function(int &f) {  
    f=f+3;  
    return f;  
}
```

```
int main() {  
    int x = 7;  
    int y;  
    y = function(x);  
    x++;  
    y--;  
    return 0;  
}
```



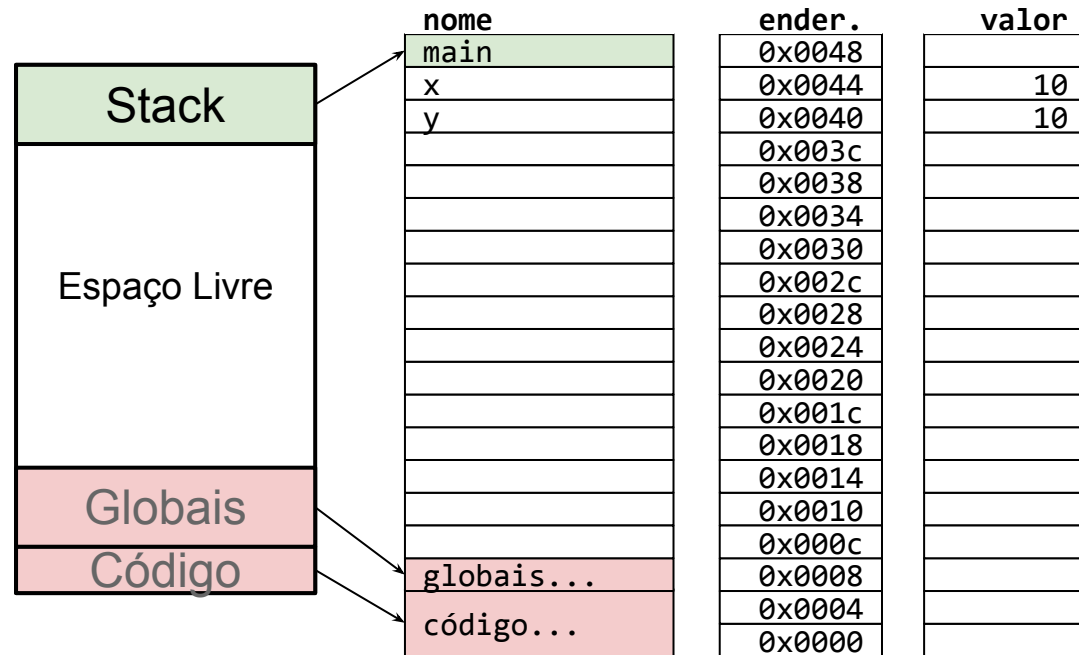
Stack

Voltamos para o main

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    int x = 7;
    int y;
    → y = function(x);
    x++;
    y--;
    return 0;
}
```



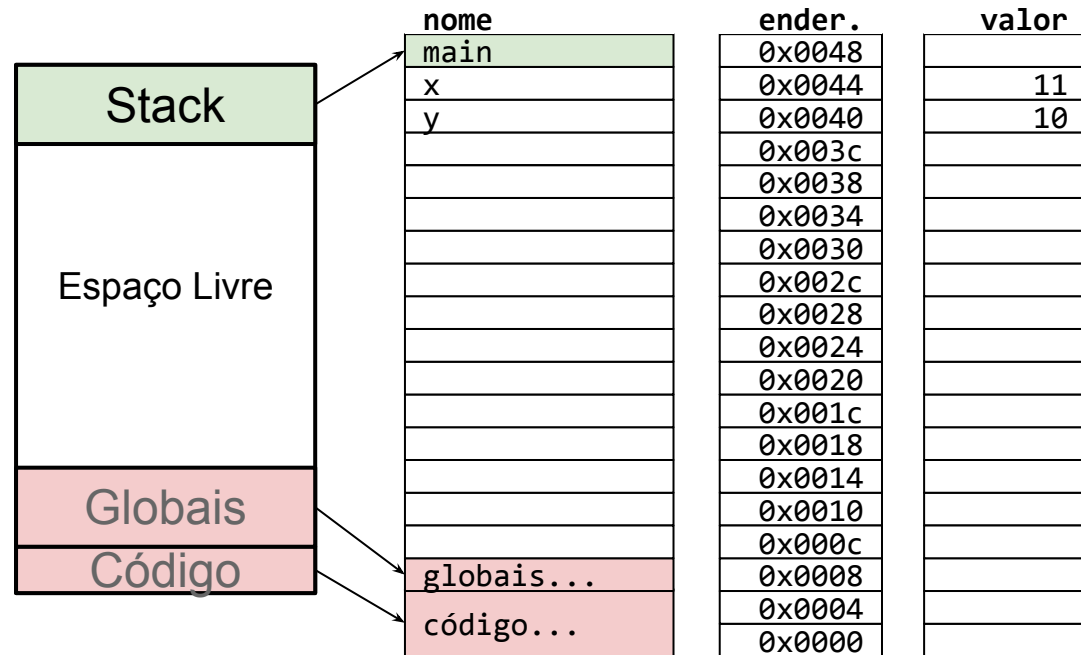
Stack

x e y estão em posições diferentes da memória

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    int x = 7;
    int y;
    y = function(x);
    x++;
    y--;
    return 0;
}
```



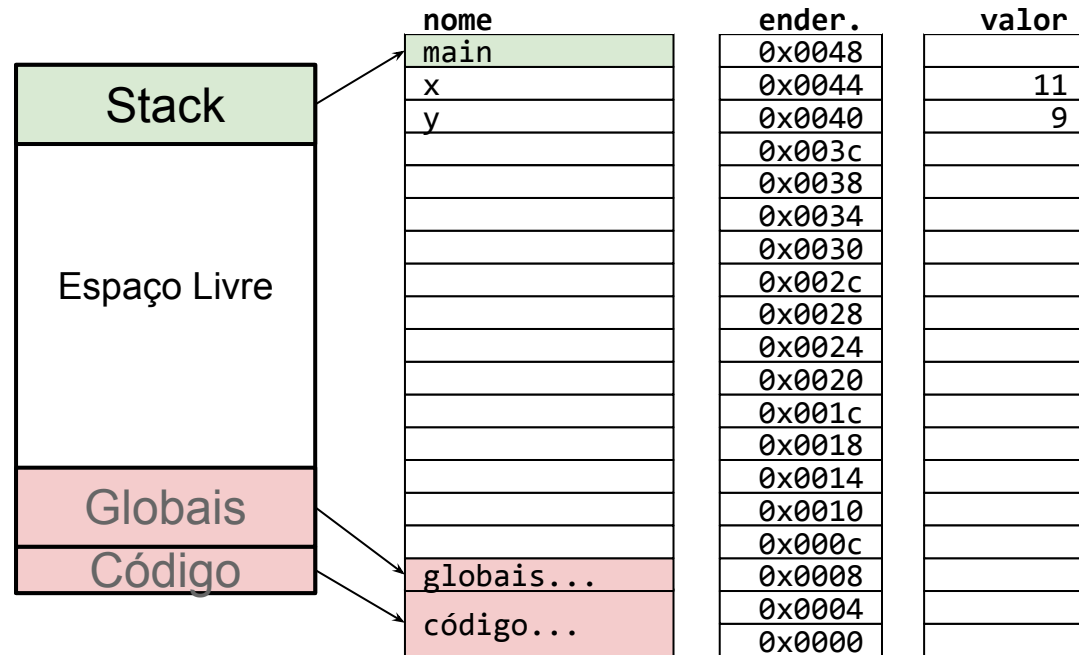
Stack

x e y estão em posições diferentes da memória

```
#include <iostream>

int function(int &f) {
    f=f+3;
    return f;
}

int main() {
    int x = 7;
    int y;
    y = function(x);
    x++;
    y--;
    return 0;
}
```



Stack/Pilha

Controla o fluxo de execução do programa

- Comportamento similar ao TAD Pilha
 - Para aqueles que viram Estrutura de Dados
- Vai guardando a posição das funções, retornos e variáveis

Referências & e ponteiros *

Quase iguais...

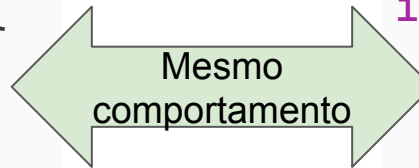
- Assim como em C: & (address of)
- Porém, podemos usar como:
- Ponteiro
 - `int *valor_ptr = &valor;`
- Referência
 - `int &valor_ptr = &valor;`

Comparando

```
#include <iostream>
```

```
int function(int &f) {  
    f=f+3;  
    return f;  
}
```

```
int main() {  
    int x = 7;  
    int y;  
    y = function(x);  
    x++;  
    y--;  
    return 0;  
}
```



```
#include <iostream>
```

```
int function(int *f) {  
    *f=*f+3;  
    return *f;  
}
```

```
int main() {  
    int x = 7;  
    int y;  
    y = function(&x);  
    x++;  
    y--;  
    return 0;  
}
```

Referências & ponteiros *

Quando usar cada um

- Ao alocar memória no heap, é comum utilizar *
 - malloc em C
 - new em C++
- Referências (&) representa um ponteiro imutável útil para evitar bugs
 - Boa prática em funções

Ainda sobre & vs *

Uma dica simples que vai lhe ajudar

- Use & sempre que possível
 - O compilador deixar passar
 - Bastante explorado em OO
 - Em funções é um bom local
- Use * sempre que necessário
 - Vetores
 - Quando cria a variável
 - Manipulação mais baixo nível (TADs)

Exemplos

Duas formas de passar por referência em C++

```
void function(int &i) {  
    int j = 10;  
    i = &j;  
}
```

```
void function(int *i) {  
    int j = 10;  
    i = &j;  
}
```

Exemplos

Duas formas de passar por referência em C++



Erro de compilação

```
void function(int &i) {  
    int j = 10;  
    i = &j;  
}
```



Compila só que é uma ótima fonte de bugs!

```
void function(int *i) {  
    int j = 10;  
    i = &j;  
}
```

Exemplo (por valor)

```
#include <iostream>

int inc(int x) {
    return ++x;
}

int main() {
    int a = 10;
    int b = inc(a);
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    std::cout << &a << std::endl;
    std::cout << &b << std::endl;
}
```


Exemplo (por referência estilo C)

```
#include <iostream>

int inc(int *x) {
    *x = *x + 1;
    return *x;
}

int main() {
    int a = 10;
    int b = inc(&a);
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    std::cout << &a << std::endl;
    std::cout << &b << std::endl;
}
```

Exemplo (por referência estilo C++)

```
#include <iostream>

int inc(int &x) {
    x++;
    return x;
}

int main() {
    int a = 10;
    int b = inc(a);
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    std::cout << &a << std::endl;
    std::cout << &b << std::endl;
}
```

Heap

Memória alocada dinamicamente

- Uma forma simples de pensar:
 - Memória da pilha é gerida automaticamente
 - Só que temos menos controle
 - Nem sempre podemos usar a pilha
- O Heap é onde mora a memória dinâmica
 - Em C utilizamos `malloc`
 - Em C++ utilizamos `new`

Heap

Liberação de memória

- O programador que libera a memória

- Em C utilizamos `free`

- Em C++ utilizamos `delete`

Exemplo

```
#include <cstdlib>
#include <iostream>

int main() {
    int *ptr_a = nullptr;
    ptr_a = new int;
    if (ptr_a == nullptr) {
        std::cout << "Memoria insuficiente!" << std::endl;
        exit(1);
    }

    std::cout << "Endereco de ptr_a: " << ptr_a << std::endl;
    *ptr_a = 90;
    std::cout << "Conteudo de ptr_a: " << *ptr_a << std::endl;
    delete ptr_a;
}
```

NULL vs nullptr

- Semanticamente igual

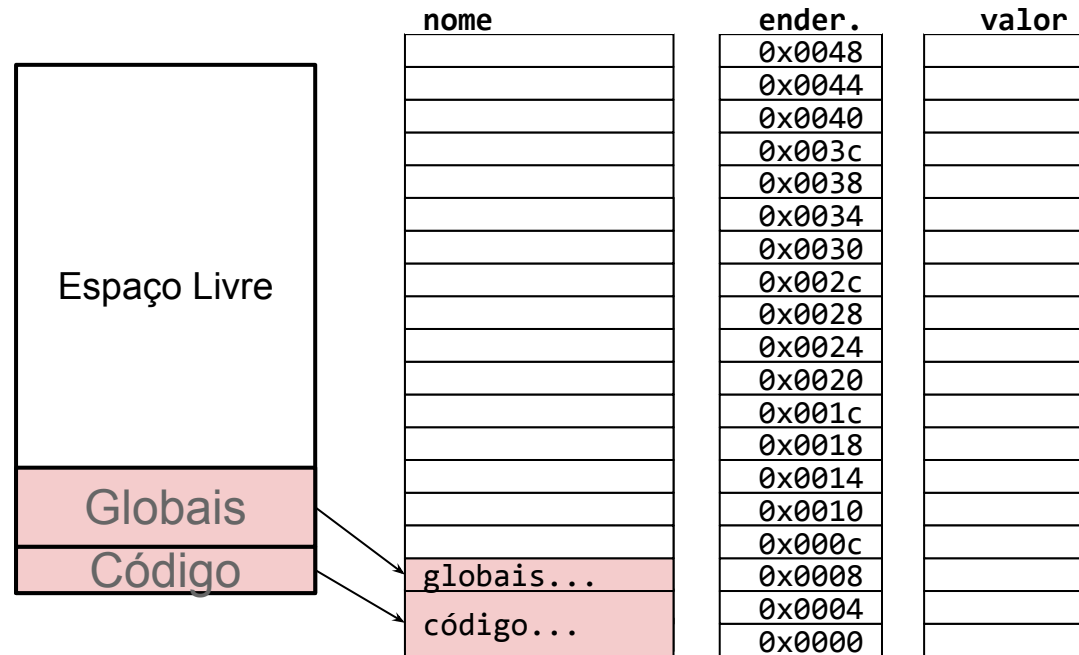
- nullptr é mais seguro

- Só pode ser atribuído para ponteiros

```
int i = NULL;           // OK, qual o valor de i?  
int i = nullptr;        // Erro de compilação  
int *p = NULL;          // OK, seguro  
int *p = nullptr;       // OK, seguro
```

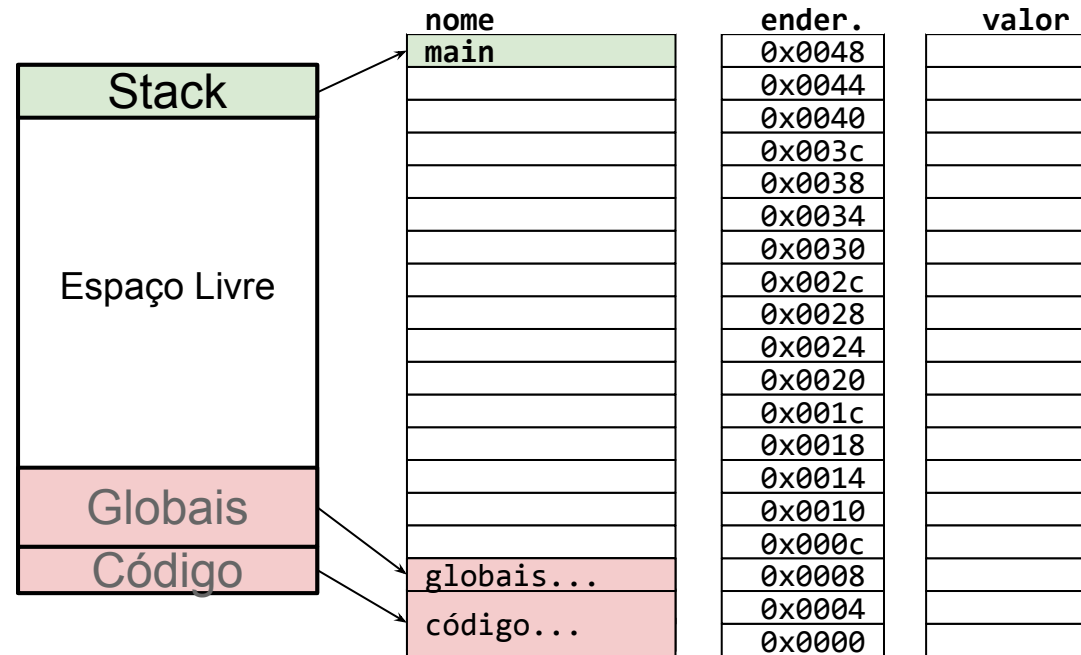
Exemplo

```
int main() {  
    int *a = new int;  
    int b = 10;  
    *a = 20;  
    a = &b;  
    *a = 30;  
}
```



Exemplo

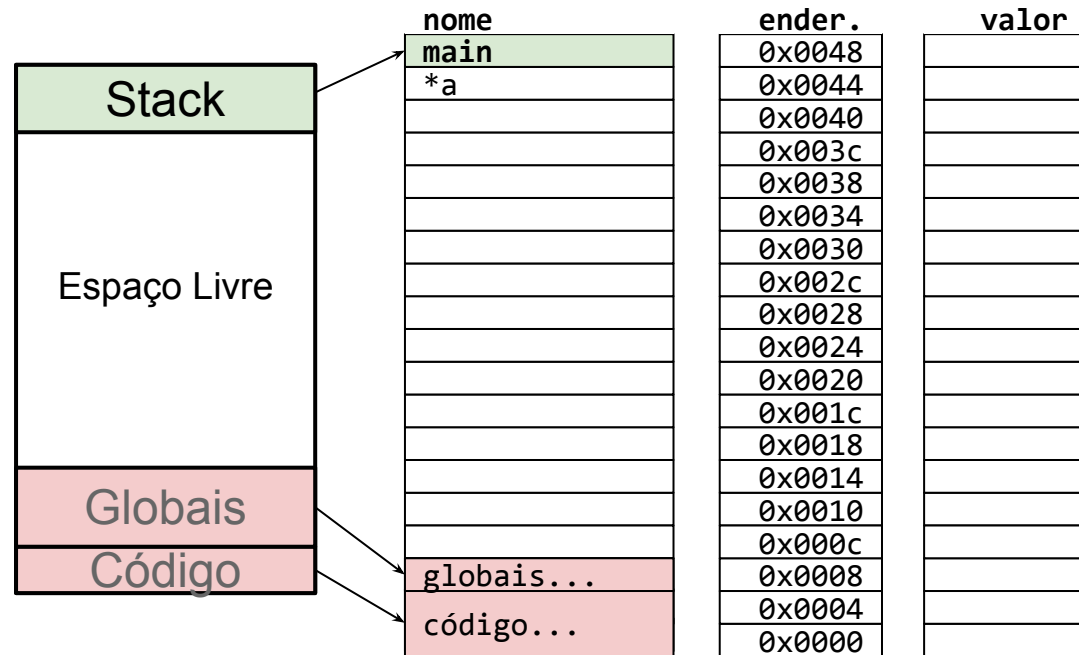
```
→ int main() {  
    int *a = new int;  
    int b = 10;  
    *a = 20;  
    a = &b;  
    *a = 30;  
}
```



Exemplo

Alocamos um inteiro no heap, a referência mora a pilha

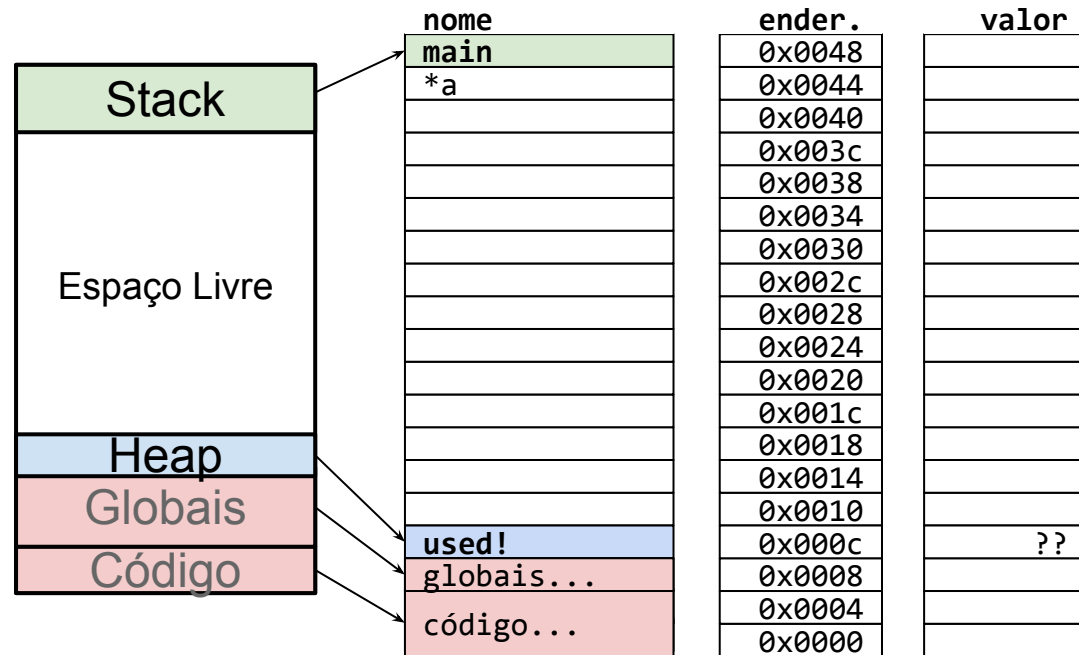
```
int main() {  
→ int *a = new int;  
  int b = 10;  
  *a = 20;  
  a = &b;  
  *a = 30;  
}
```



Exemplo

Alocamos um inteiro no heap, a referência mora a pilha

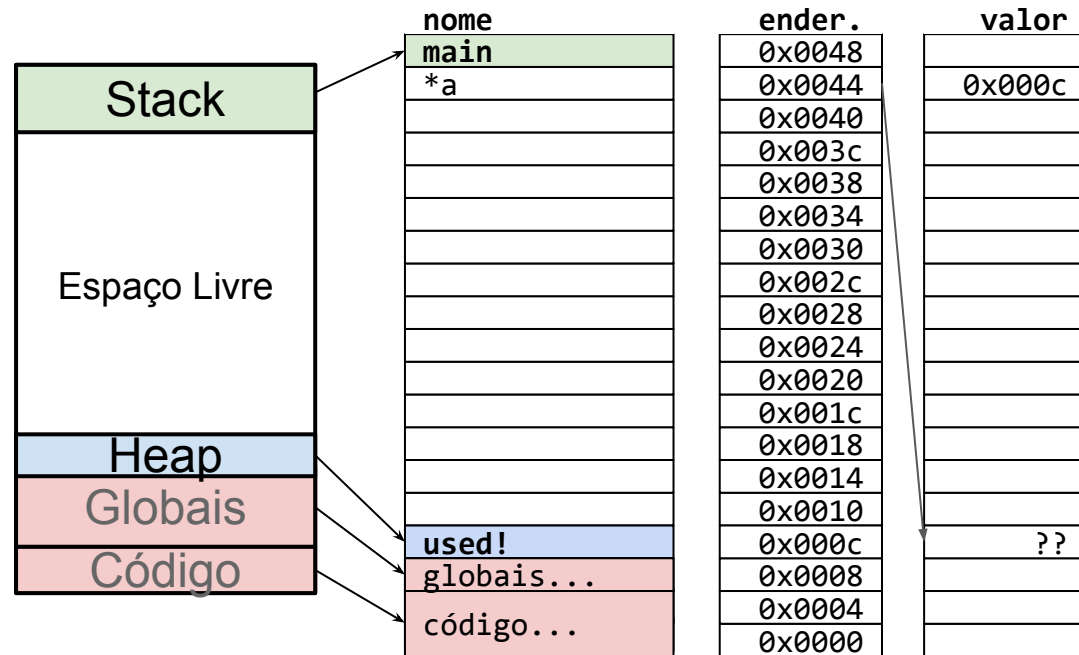
```
int main() {  
→ int *a = new int;  
  int b = 10;  
  *a = 20;  
  a = &b;  
  *a = 30;  
}
```



Exemplo

Alocamos um inteiro no heap, a referência mora a pilha

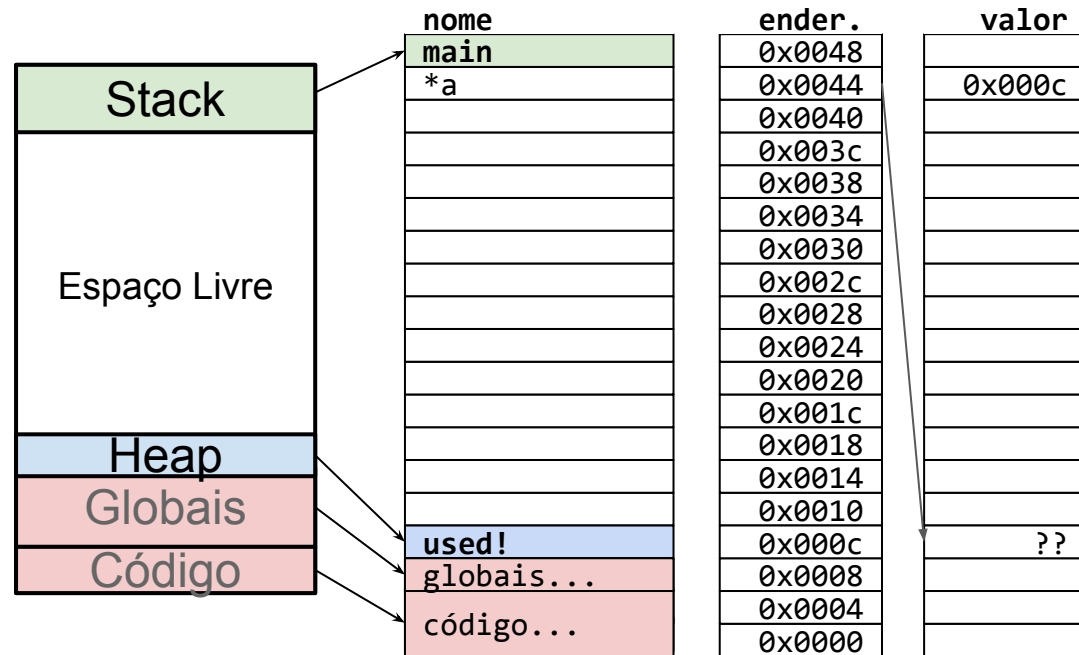
```
int main() {  
→ int *a = new int;  
  int b = 10;  
  *a = 20;  
  a = &b;  
  *a = 30;  
}
```



Exemplo

Não temos variável no heap! Acesso por *a

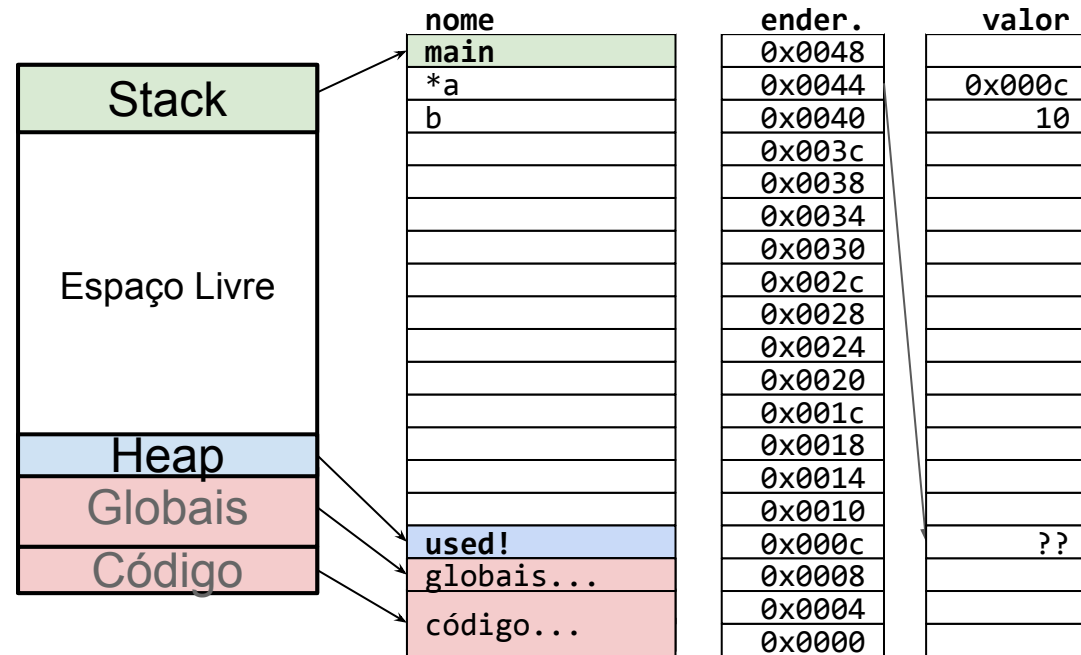
```
int main() {  
→ int *a = new int;  
  int b = 10;  
  *a = 20;  
  a = &b;  
  *a = 30;  
}
```



Exemplo

b mora na pilha

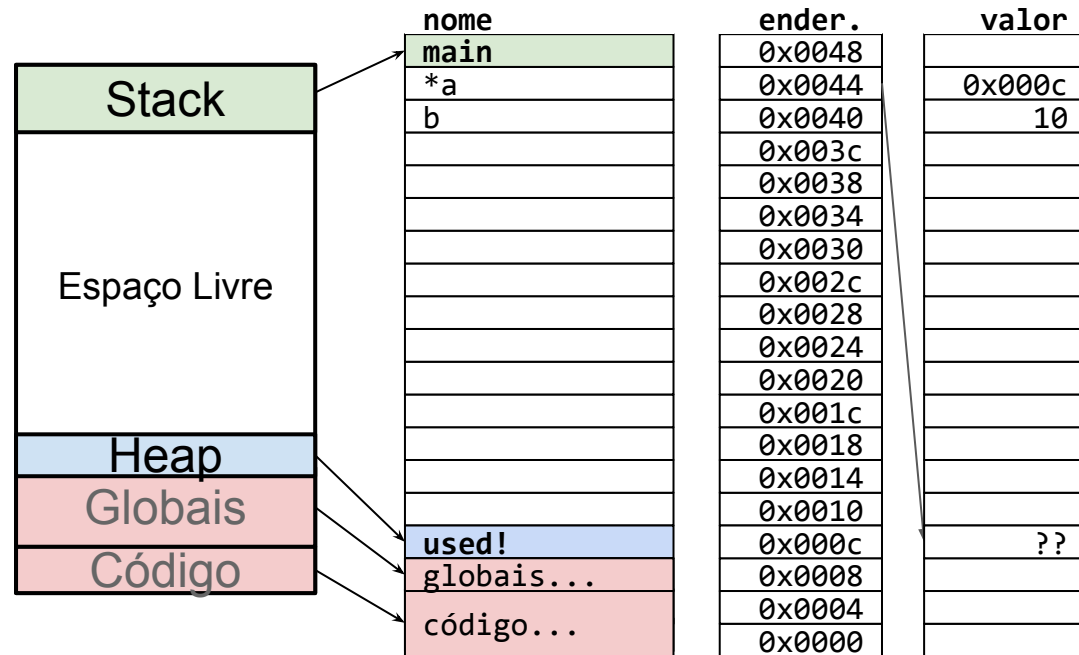
```
int main() {  
    int *a = new int;  
    → int b = 10;  
    *a = 20;  
    a = &b;  
    *a = 30;  
}
```



Exemplo

Como que o diagrama muda?

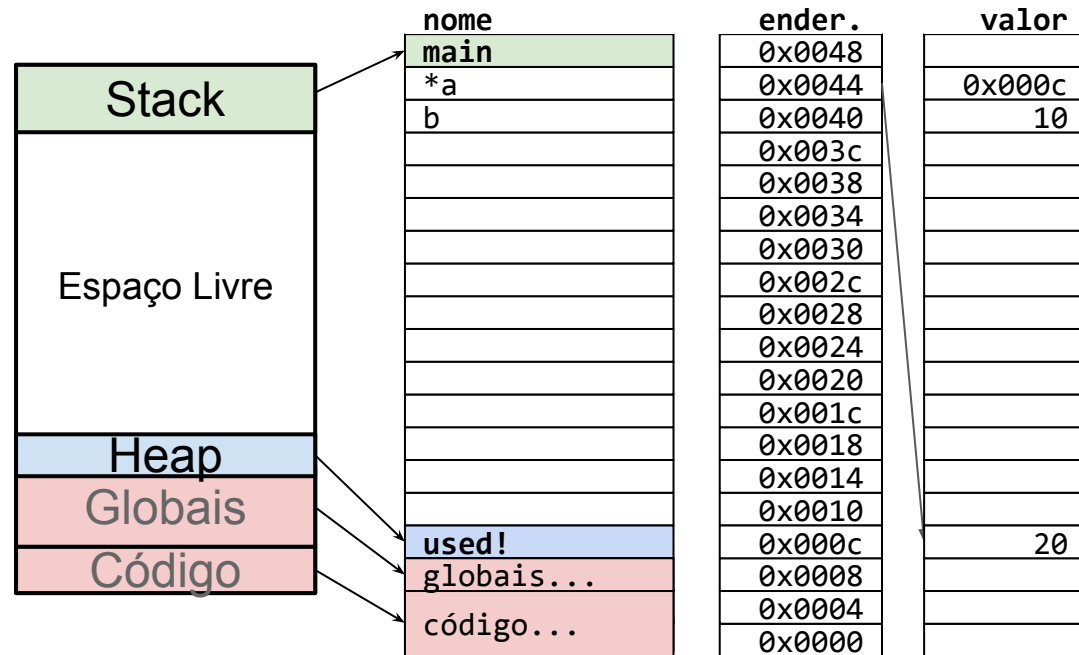
```
int main() {  
    int *a = new int;  
    int b = 10;  
    → *a = 20;  
    a = &b;  
    *a = 30;  
}
```



Exemplo

Valor do local de memória referenciado por a = 20

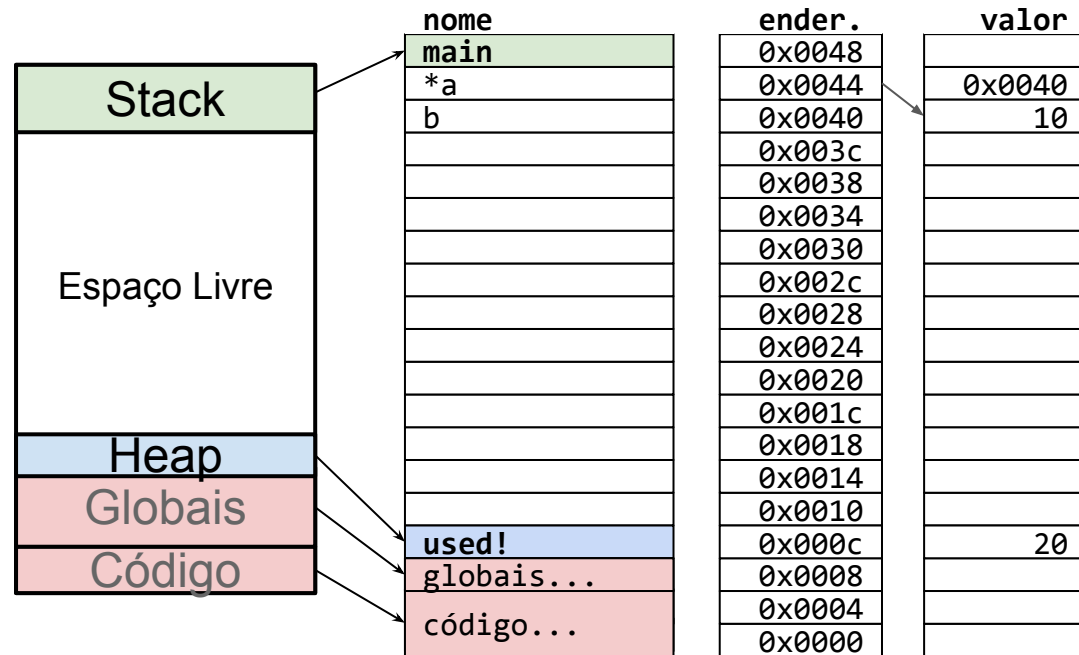
```
int main() {  
    int *a = new int;  
    int b = 10;  
    → *a = 20;  
    a = &b;  
    *a = 30;  
}
```



Exemplo

Como que o diagrama muda?

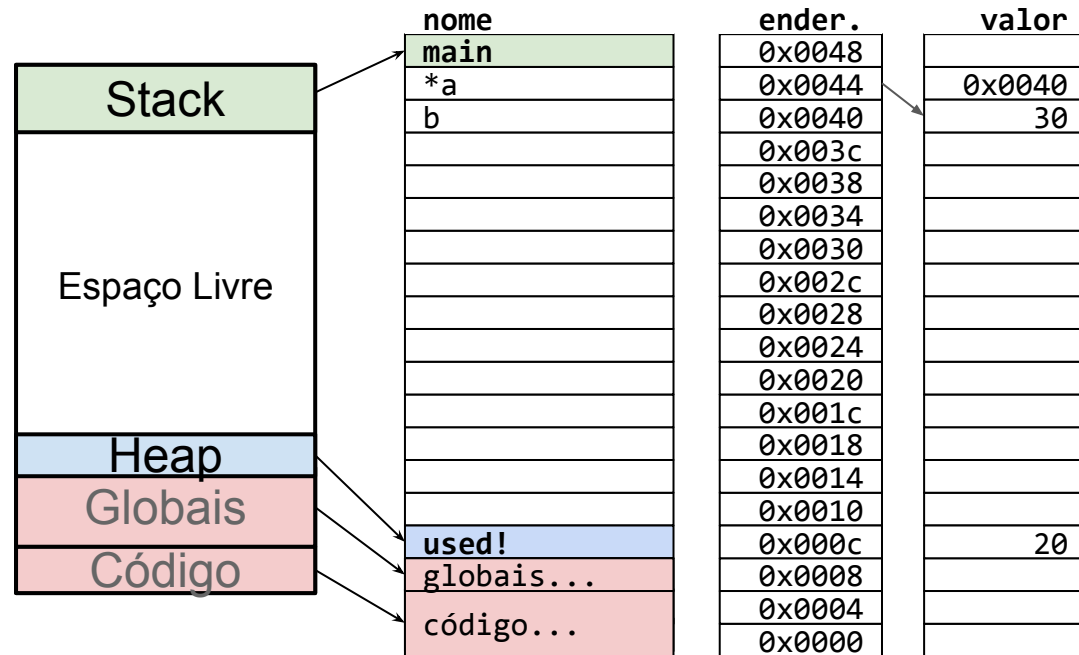
```
int main() {  
    int *a = new int;  
    int b = 10;  
    *a = 20;  
    → a = &b;  
    *a = 30;  
}
```



Exemplo

Como que o diagrama muda?

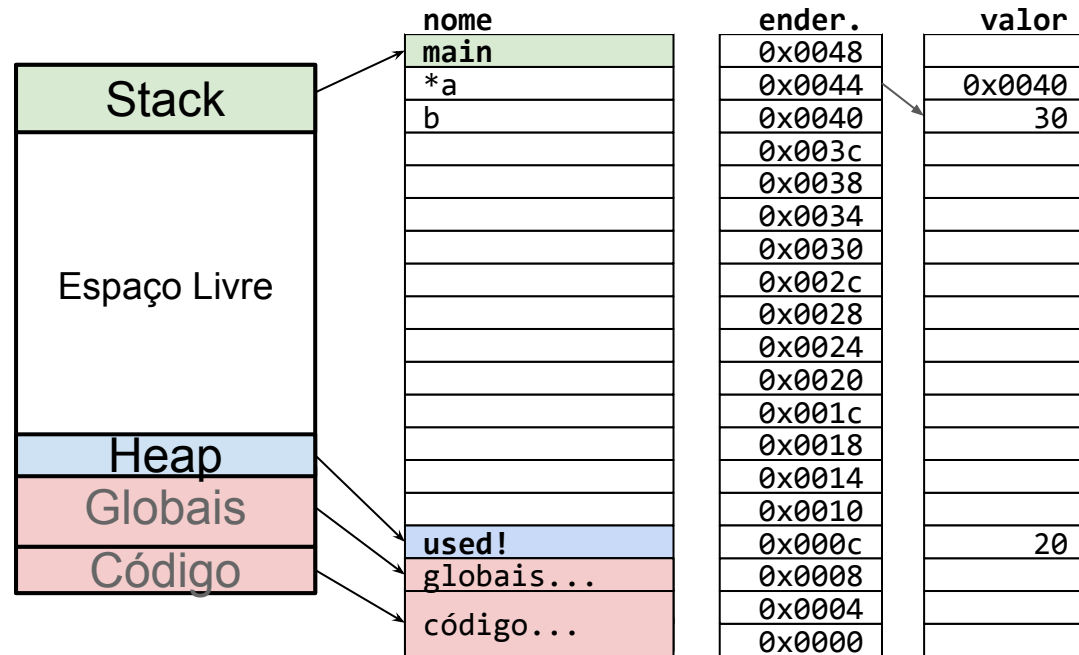
```
int main() {  
    int *a = new int;  
    int b = 10;  
    *a = 20;  
    a = &b;  
    → *a = 30;  
}
```



Exemplo

Qual o problema?

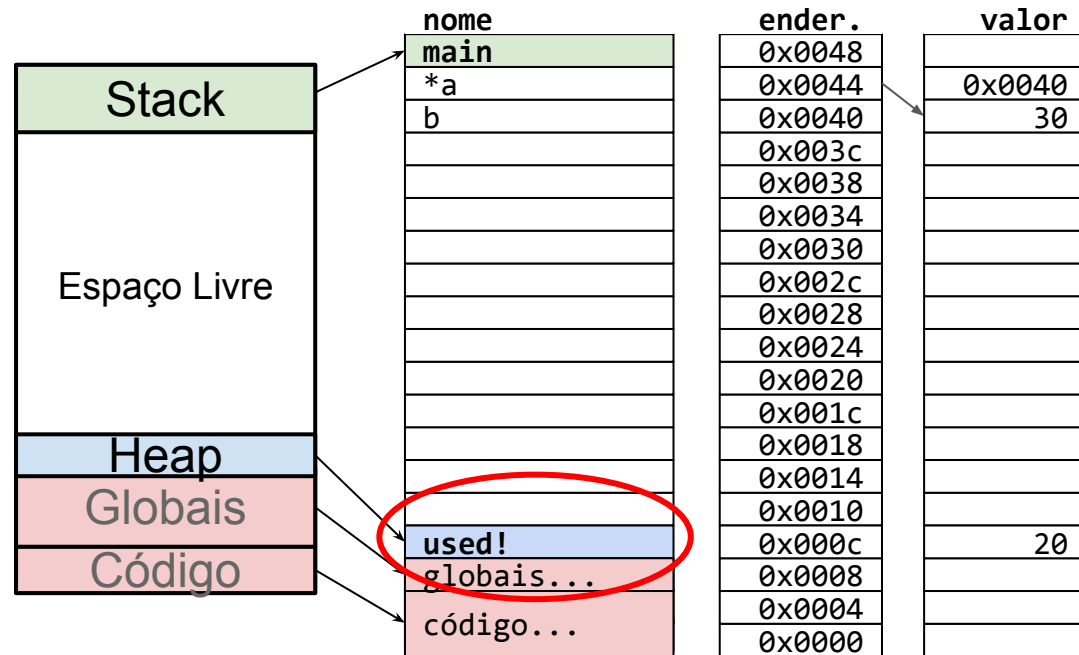
```
int main() {  
    int *a = new int;  
    int b = 10;  
    *a = 20;  
    a = &b;  
    → *a = 30;  
}
```



Exemplo

Qual o problema?

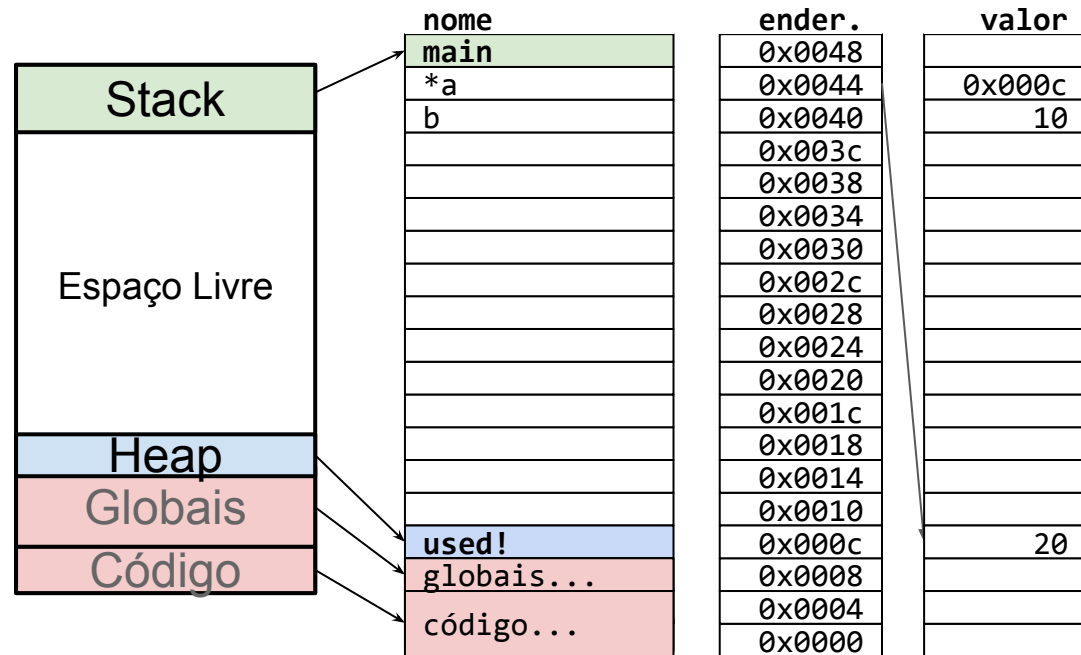
```
int main() {  
    int *a = new int;  
    int b = 10;  
    *a = 20;  
    a = &b;  
    → *a = 30;  
}
```



Exemplo

Qual o problema?

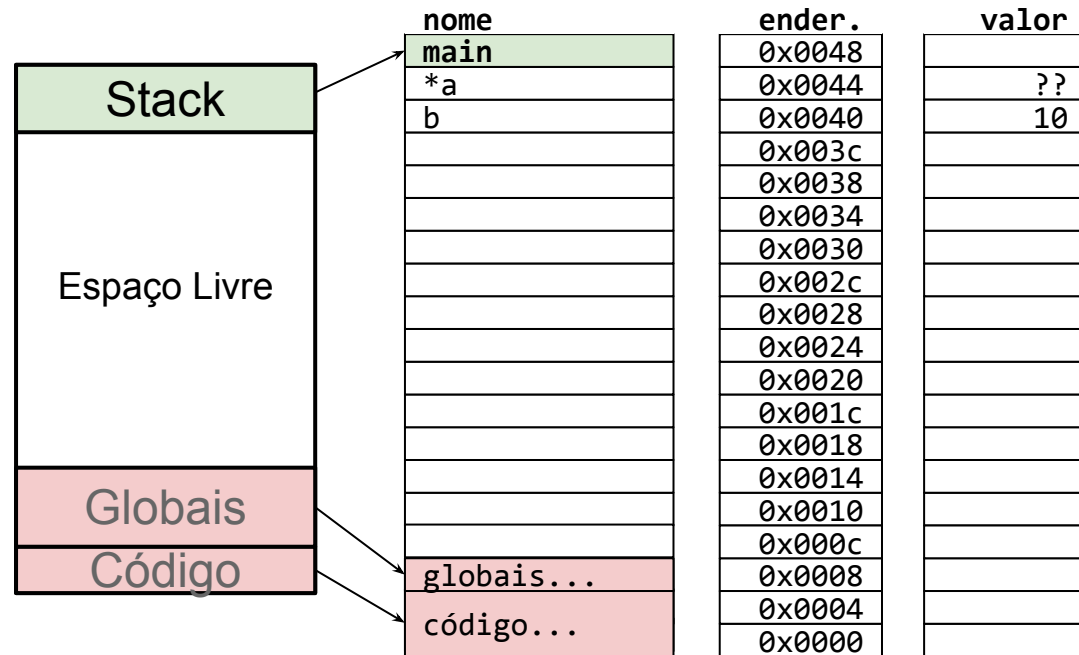
```
int main() {  
    int *a = new int;  
    int b = 10;  
    *a = 20;  
    → delete a;  
    a = &b;  
    *a = 30;  
}
```



Exemplo

Qual o problema?

```
int main() {  
    int *a = new int;  
    int b = 10;  
    *a = 20;  
    delete a;  
    → a = &b;  
    *a = 30;  
}
```



Alocação dinâmica de vetores

- Normalmente, a alocação dinâmica é utilizada para criar vetores em tempo de execução
- Exemplo:

```
int *p = new int[10];
```

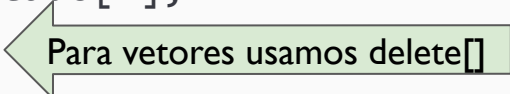
- Aloca um vetor de inteiros com 10 posições. A manipulação é feita normalmente: $p[i] = \dots$
- O apontador p guarda o endereço (aponta) da primeira posição do vetor.

Exemplo vetores

```
#include <cstdlib>
#include <iostream>

int *produto_interno(int n, int *vetor_a, int *vetor_b) {
    int *resultado = new int[n];
    for (int i = 0; i < n; i++)
        resultado[i] = vetor_a[i] * vetor_b[i];
    return resultado;
}

int main() {
    int vetor_a[3] = {1, 2, 3};
    int vetor_b[3] = {3, 2, 1};
    int *resultado = produto_interno(3, vetor_a, vetor_b);
    for (int i = 0; i < 3; i++)
        std::cout << resultado[i];
    delete[] resultado;
}
```

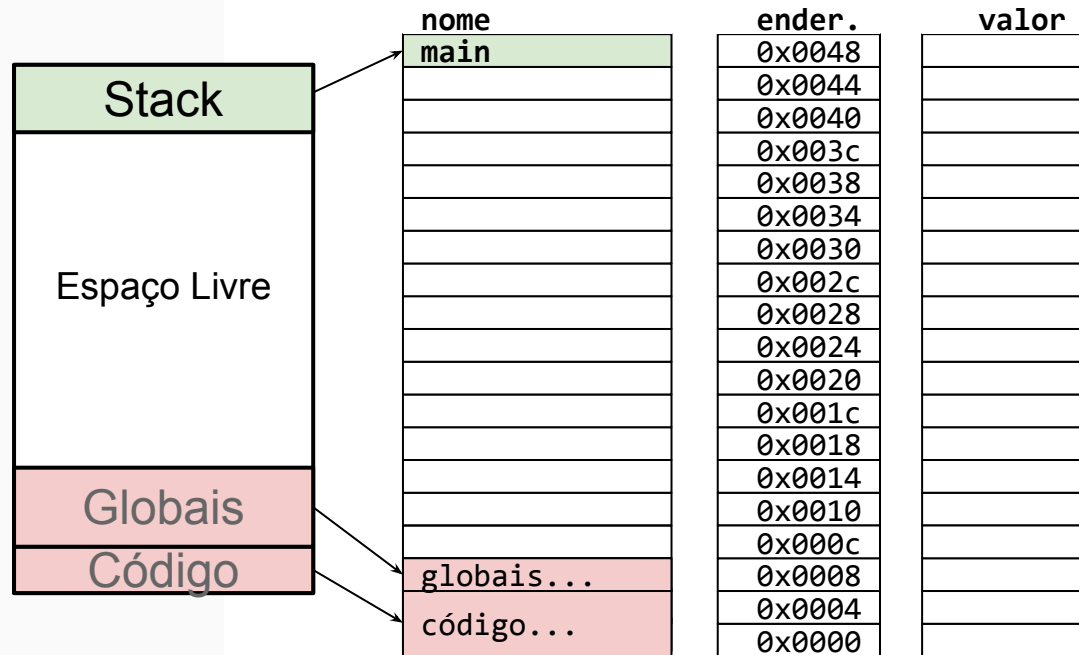


Para vetores usamos delete[]

Exemplo vetores

```
int *produto(int n, int *va, int *vb) {  
    int *resultado = new int[n];  
    for (int i = 0; i < n; i++)  
        resultado[i] = va[i] * vb[i];  
    return resultado;  
}
```

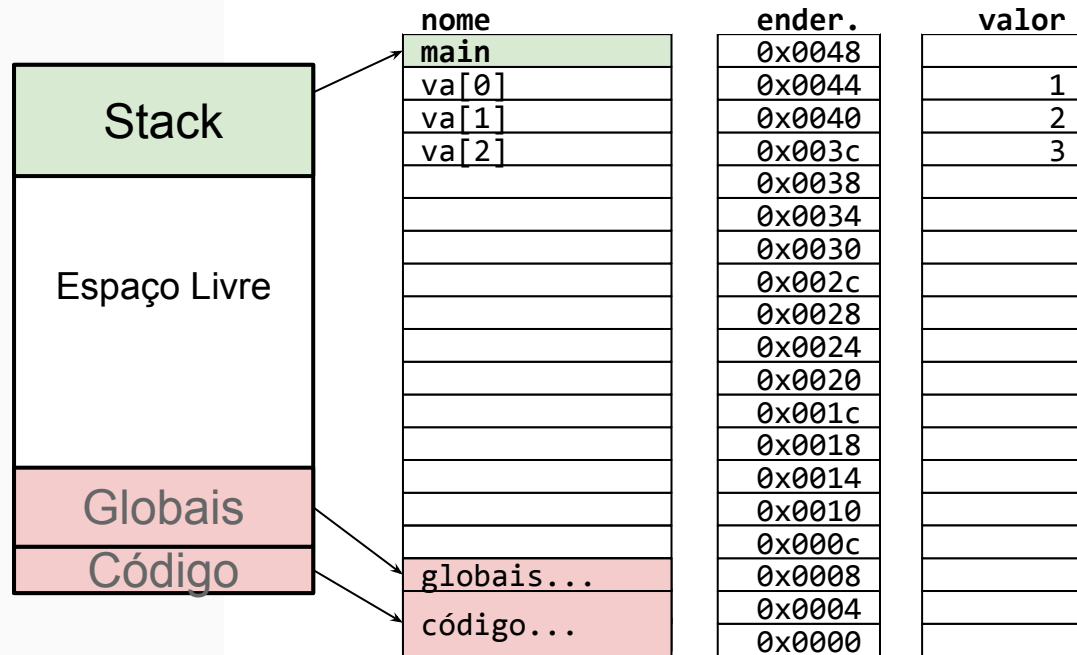
```
int main() {  
→ int va[3] = {1, 2, 3};  
  int vb[3] = {3, 2, 1};  
  int *resultado = nullptr;  
  resultado = produto(3, va, vb);  
  for (int i = 0; i < 3; i++)  
      std::cout << resultado[i];  
  delete[] resultado;  
}
```



Exemplo vetores

```
int *produto(int n, int *va, int *vb) {  
    int *resultado = new int[n];  
    for (int i = 0; i < n; i++)  
        resultado[i] = va[i] * vb[i];  
    return resultado;  
}
```

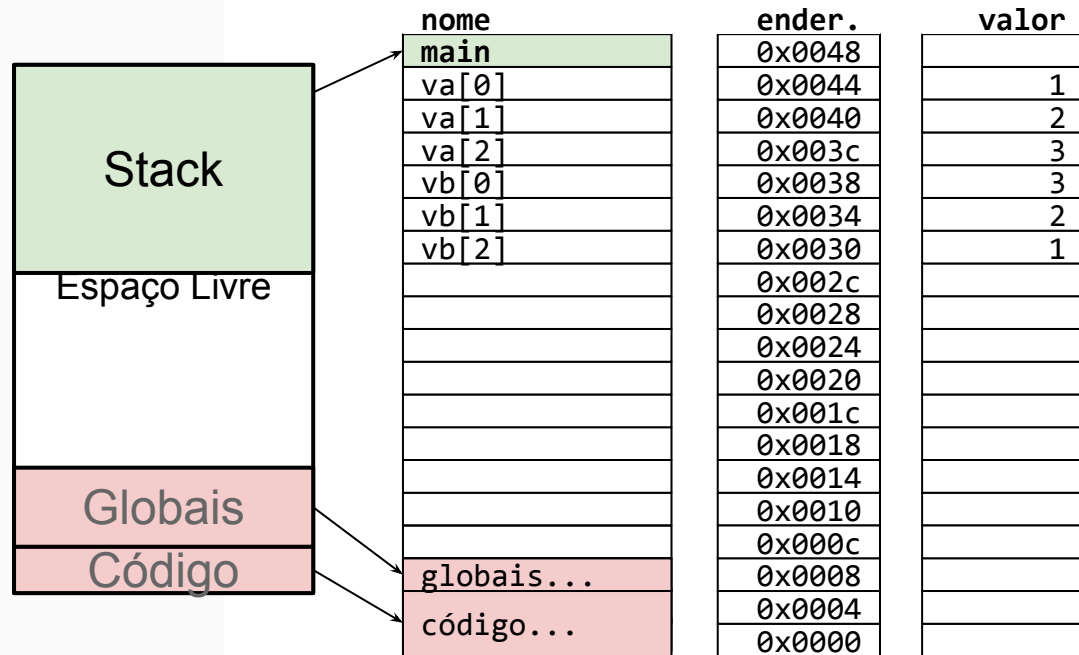
```
int main() {  
    int va[3] = {1, 2, 3};  
    int vb[3] = {3, 2, 1};  
    int *resultado = nullptr;  
    resultado = produto(3, va, vb);  
    for (int i = 0; i < 3; i++)  
        std::cout << resultado[i];  
    delete[] resultado;  
}
```



Exemplo vetores

```
int *produto(int n, int *va, int *vb) {  
    int *resultado = new int[n];  
    for (int i = 0; i < n; i++)  
        resultado[i] = va[i] * vb[i];  
    return resultado;  
}
```

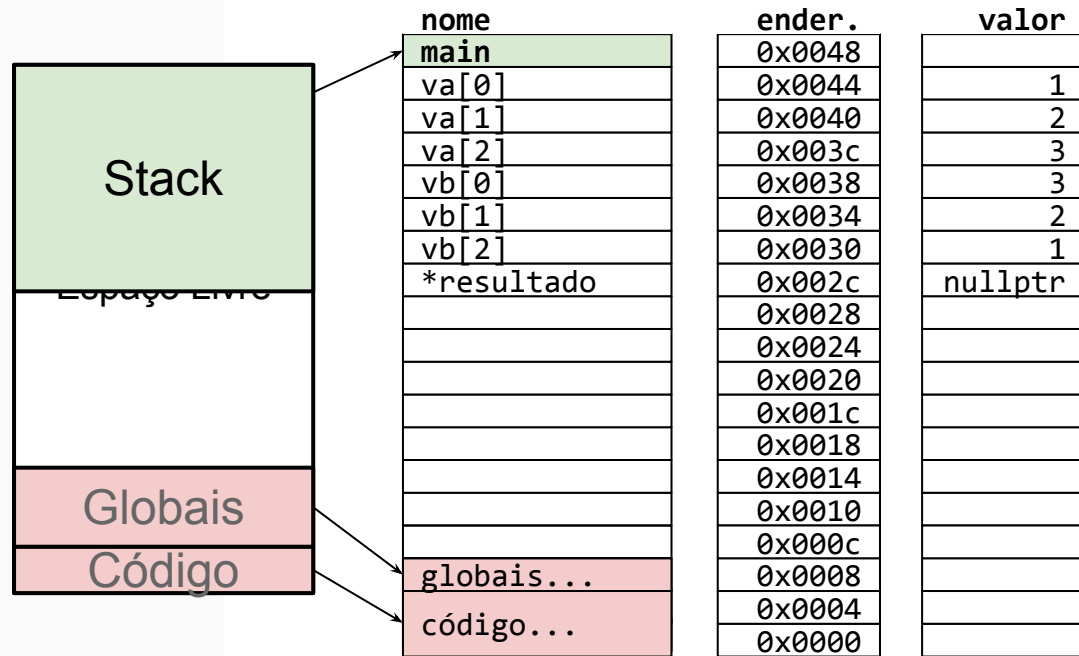
```
int main() {  
    int va[3] = {1, 2, 3};  
    int vb[3] = {3, 2, 1};  
    int *resultado = nullptr;  
    resultado = produto(3, va, vb);  
    for (int i = 0; i < 3; i++)  
        std::cout << resultado[i];  
    delete[] resultado;  
}
```



Exemplo vetores

```
int *produto(int n, int *va, int *vb) {  
    int *resultado = new int[n];  
    for (int i = 0; i < n; i++)  
        resultado[i] = va[i] * vb[i];  
    return resultado;  
}
```

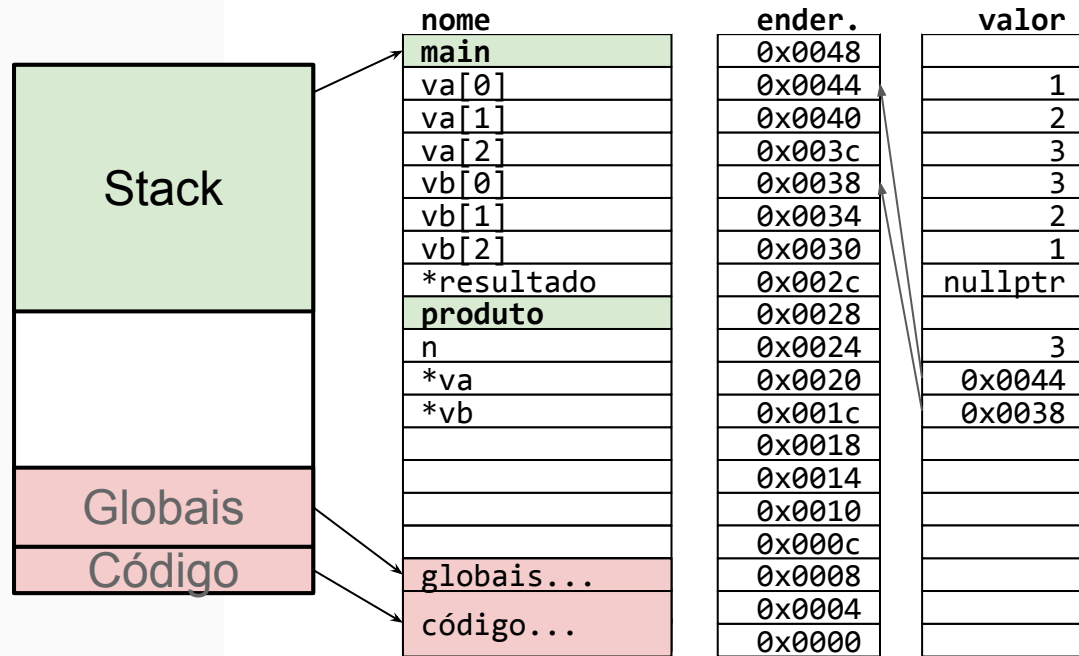
```
int main() {  
    int va[3] = {1, 2, 3};  
    int vb[3] = {3, 2, 1};  
    int *resultado = nullptr;  
    resultado = produto(3, va, vb);  
    for (int i = 0; i < 3; i++)  
        std::cout << resultado[i];  
    delete[] resultado;  
}
```



Exemplo vetores

```
int *produto(int n, int *va, int *vb) {  
    int *resultado = new int[n];  
    for (int i = 0; i < n; i++)  
        resultado[i] = va[i] * vb[i];  
    return resultado;  
}
```

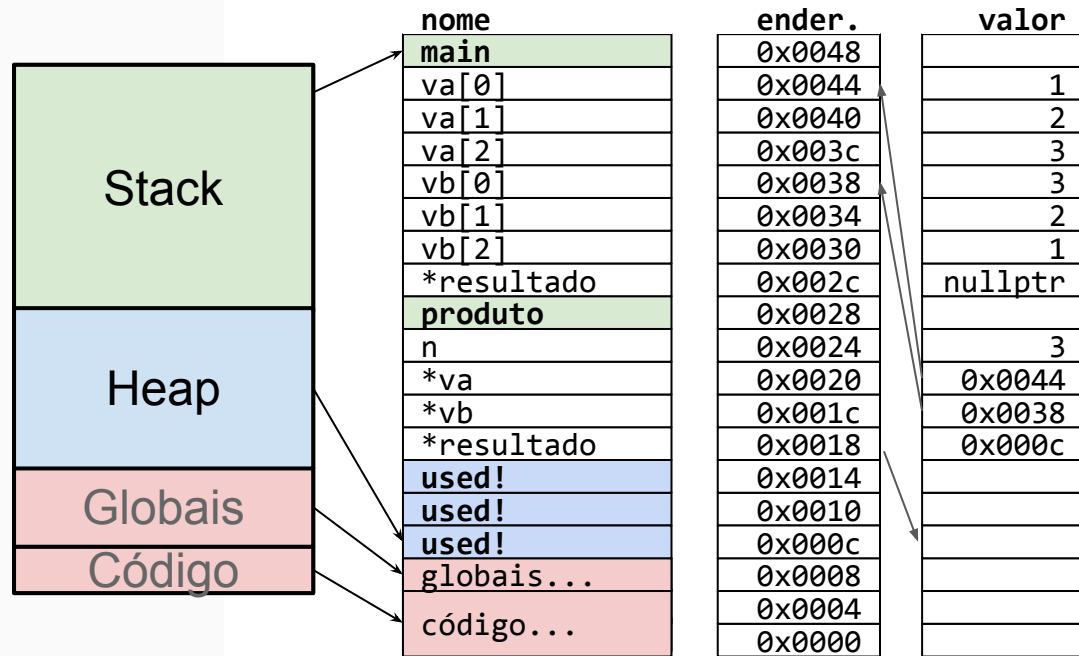
```
int main() {  
    int va[3] = {1, 2, 3};  
    int vb[3] = {3, 2, 1};  
    int *resultado = nullptr;  
    resultado = produto(3, va, vb);  
    for (int i = 0; i < 3; i++)  
        std::cout << resultado[i];  
    delete[] resultado;  
}
```



Exemplo vetores

```
int *produto(int n, int *va, int *vb) {  
    int *resultado = new int[n];  
    for (int i = 0; i < n; i++)  
        resultado[i] = va[i] * vb[i];  
    return resultado;  
}
```

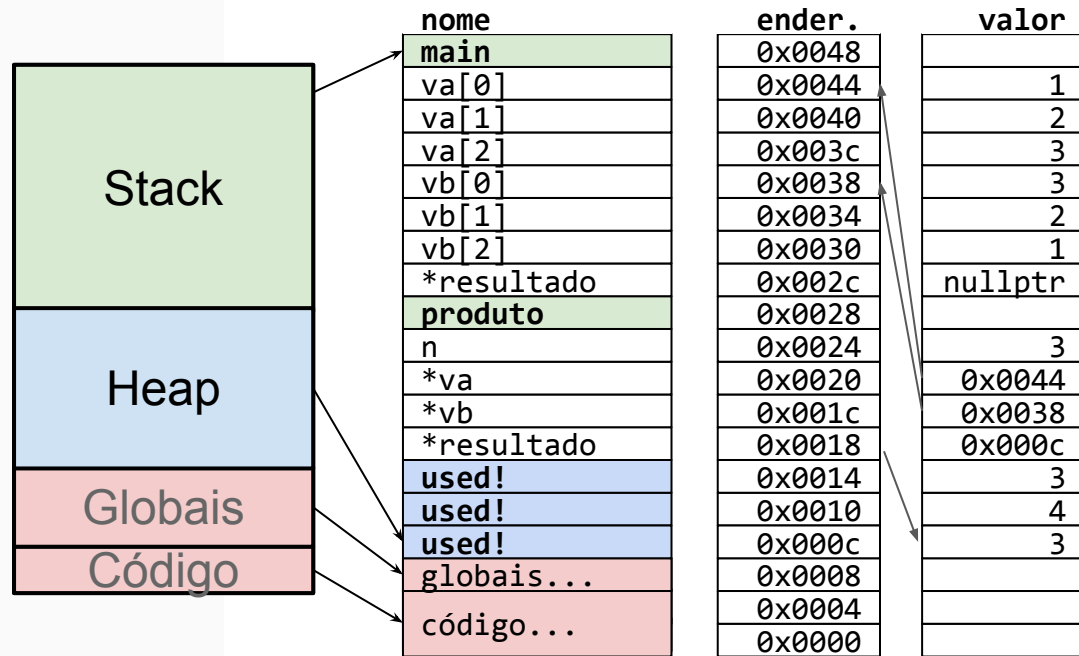
```
int main() {  
    int va[3] = {1, 2, 3};  
    int vb[3] = {3, 2, 1};  
    int *resultado = nullptr;  
    resultado = produto(3, va, vb);  
    for (int i = 0; i < 3; i++)  
        std::cout << resultado[i];  
    delete[] resultado;  
}
```



Exemplo vetores

```
int *produto(int n, int *va, int *vb) {  
    int *resultado = new int[n];  
    for (int i = 0; i < n; i++)  
        resultado[i] = va[i] * vb[i];  
    return resultado;  
}
```

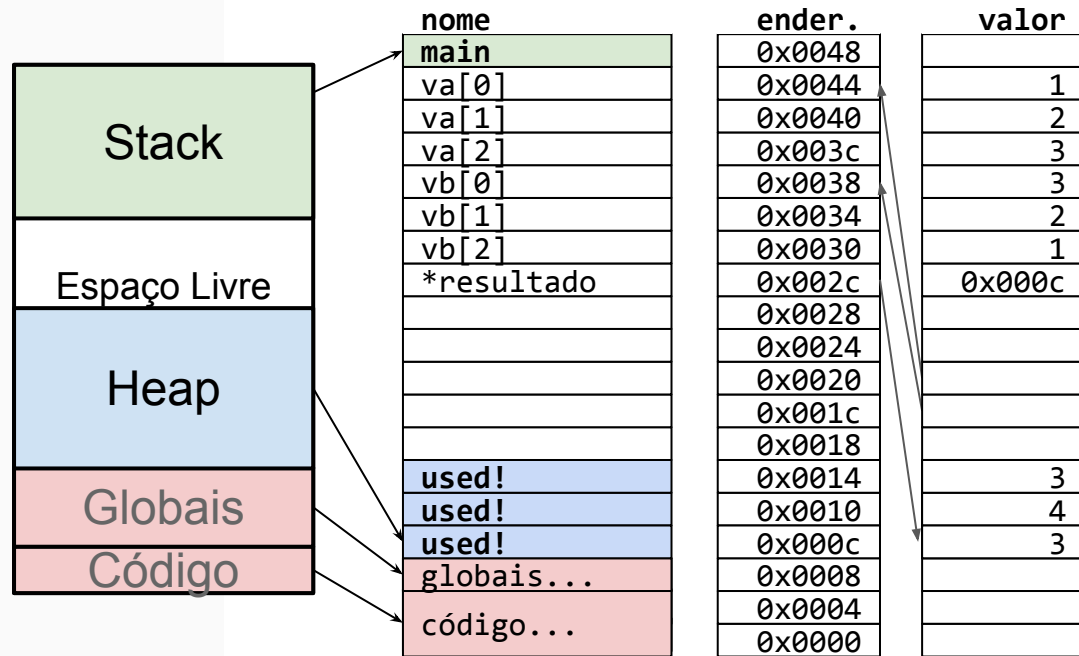
```
int main() {  
    int va[3] = {1, 2, 3};  
    int vb[3] = {3, 2, 1};  
    int *resultado = nullptr;  
    resultado = produto(3, va, vb);  
    for (int i = 0; i < 3; i++)  
        std::cout << resultado[i];  
    delete[] resultado;  
}
```



Exemplo vetores

```
int *produto(int n, int *va, int *vb) {  
    int *resultado = new int[n];  
    for (int i = 0; i < n; i++)  
        resultado[i] = va[i] * vb[i];  
    return resultado;  
}
```

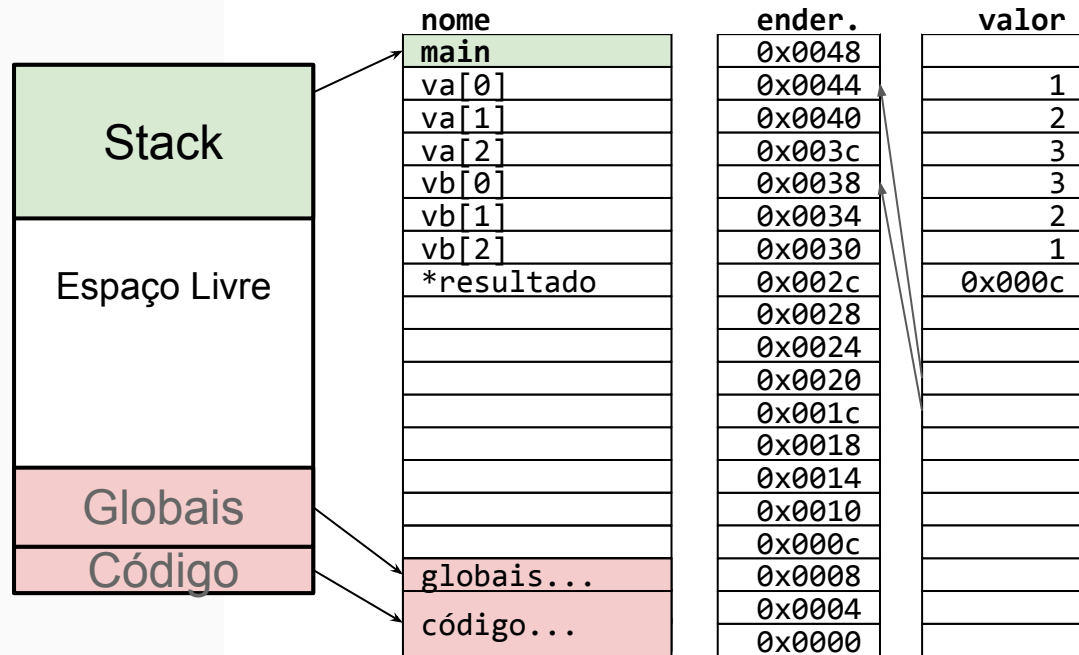
```
int main() {  
    int va[3] = {1, 2, 3};  
    int vb[3] = {3, 2, 1};  
    int *resultado = nullptr;  
    resultado = produto(3, va, vb);  
    for (int i = 0; i < 3; i++)  
        std::cout << resultado[i];  
    delete[] resultado;  
}
```



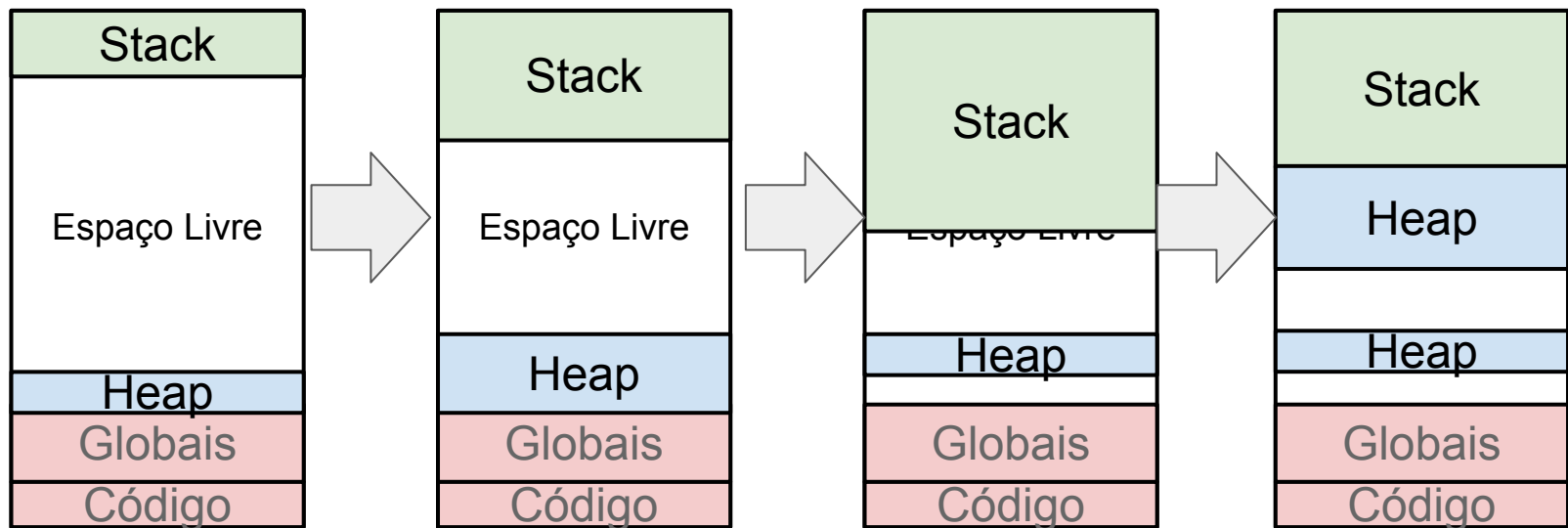
Exemplo vetores

```
int *produto(int n, int *va, int *vb) {  
    int *resultado = new int[n];  
    for (int i = 0; i < n; i++)  
        resultado[i] = va[i] * vb[i];  
    return resultado;  
}
```

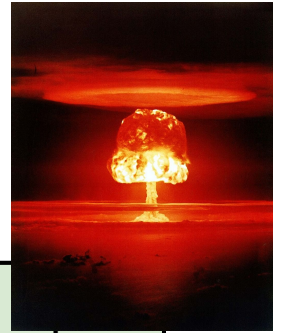
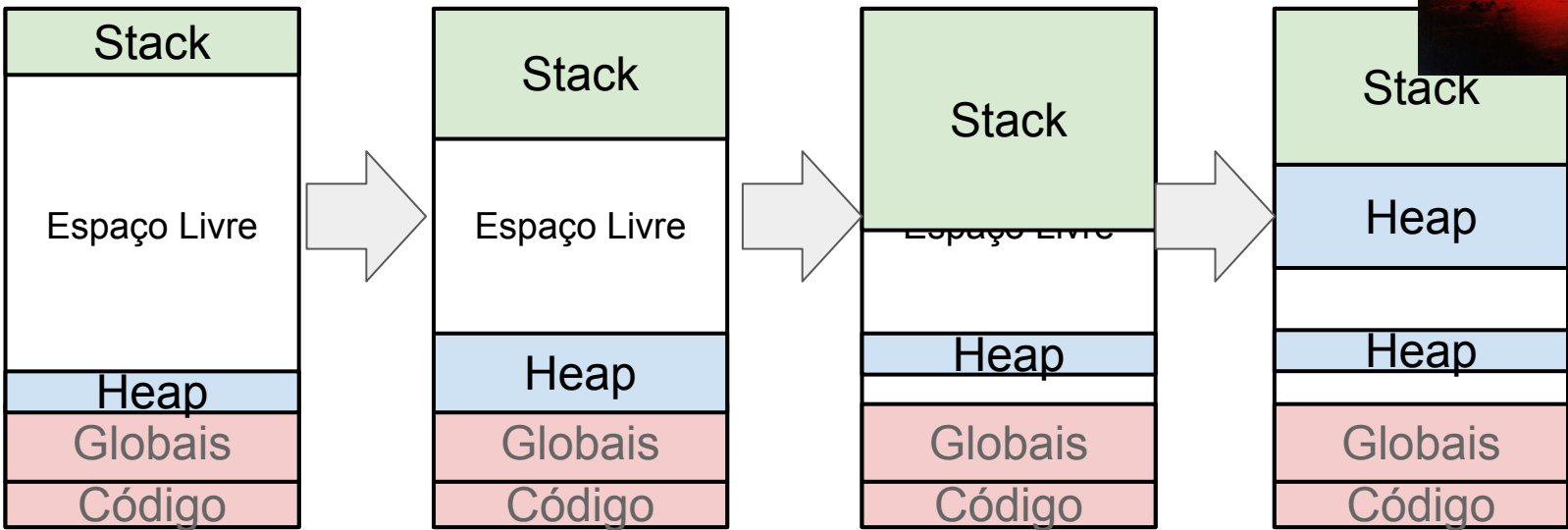
```
int main() {  
    int va[3] = {1, 2, 3};  
    int vb[3] = {3, 2, 1};  
    int *resultado = nullptr;  
    resultado = produto(3, va, vb);  
    for (int i = 0; i < 3; i++)  
        std::cout << resultado[i];  
    delete[] resultado;  
}
```



Sempre libere o heap



Sempre libere o heap



Erros comuns

- Esquecer de alocar memória e tentar acessar o conteúdo da variável
- Copiar o valor do apontador ao invés do valor da variável apontada
- Esquecer de desalocar memória
 - Ela é desalocada ao fim do programa ou procedimento função onde a variável está declarada, mas pode ser um problema em loops
- Tentar acessar o conteúdo da variável depois de desalocá-la