Ponteiros e organização de memória

*aula baseada nos slides do prof. Flávio Vinícius

Relembrando de Ponteiros

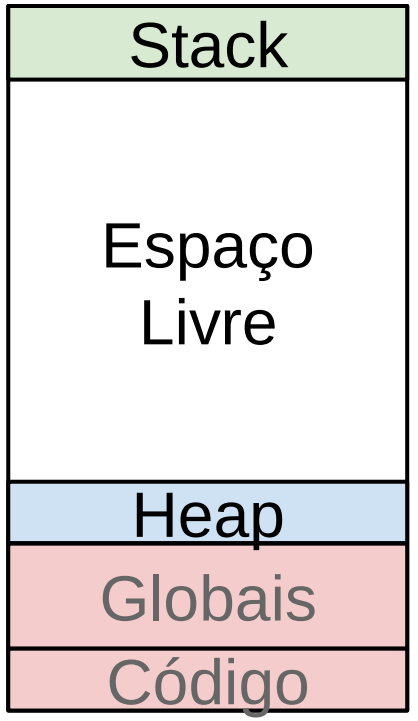
& → retorne o endereço de memória de uma variável. Em outras palavras, um ponteiro

* → utilizando para indicar ponteiros

```
// y é uma variável com valor 10
// y = 10; y++ = 11;
int  y = 10;

// y_ptr uma variável com o valor do endereço de y
// y_ptr = 0x7fff6b481f90; y_ptr++ = 0x7fff6b481f94;
int *y_ptr = &y;
```

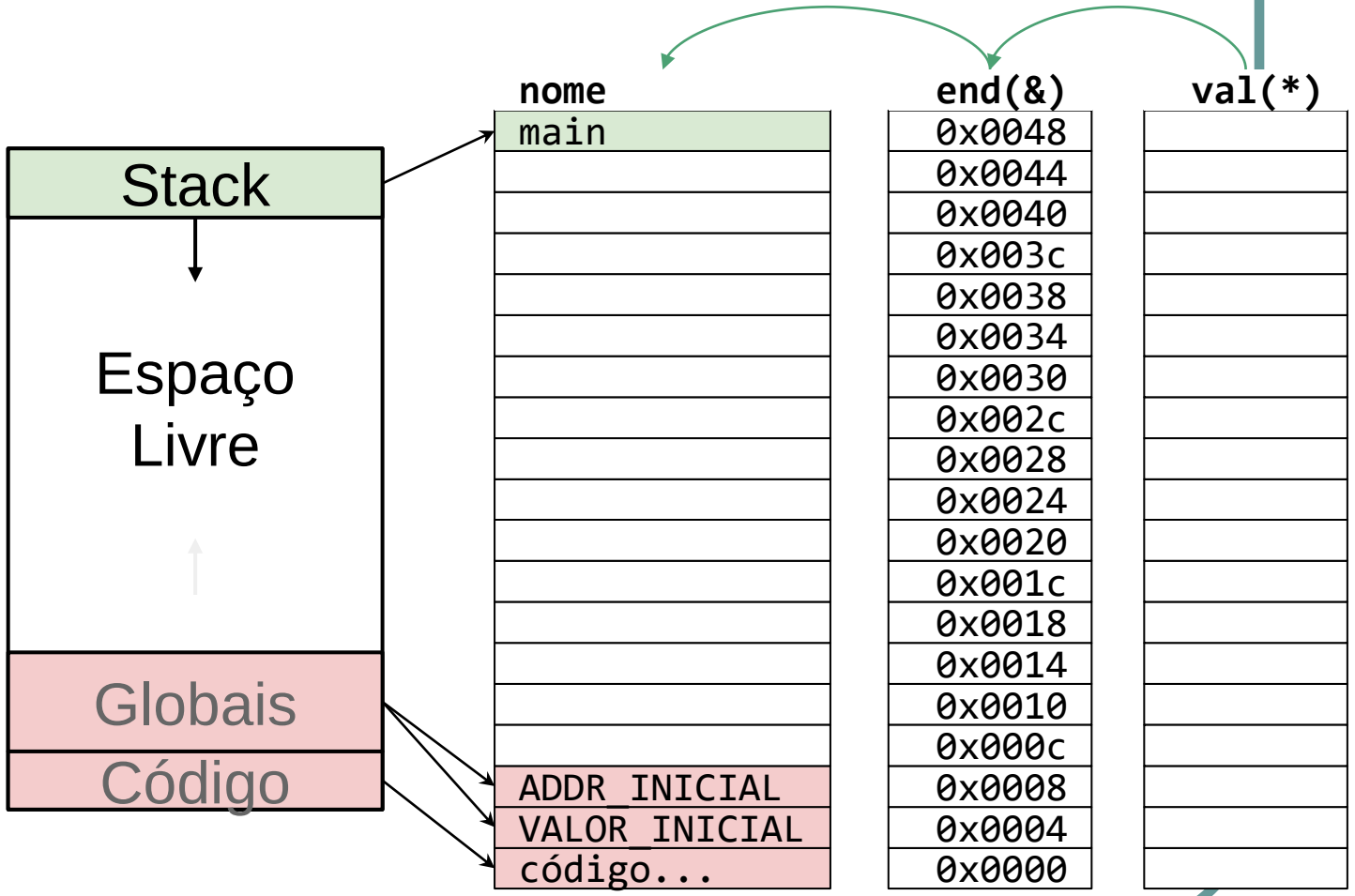
Esquema de um Programa na Memória

- Podemos simplificar um programa com uma esquema como o abaixo
- [Sempre] Código
  - Que ocupa espaço
- [Pode ter] Variáveis globais
  - Que ocupa espaço
- [Sempre] Stack ou Pilha
  - Utilizado pelo hardware para chamar funções
- [Sempre] Heap
  - Alocação dinâmica (esta aula)



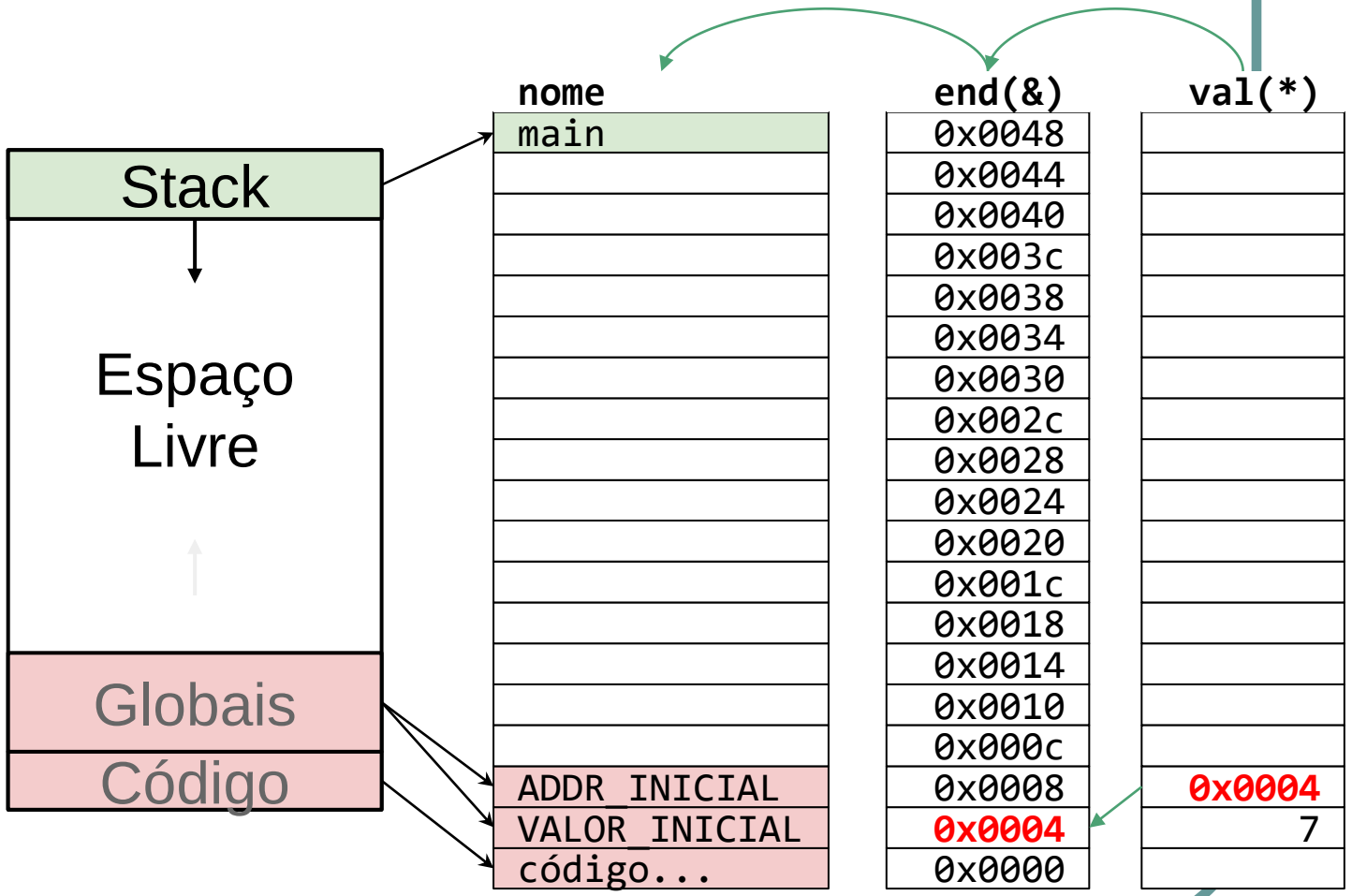
Exemplo

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



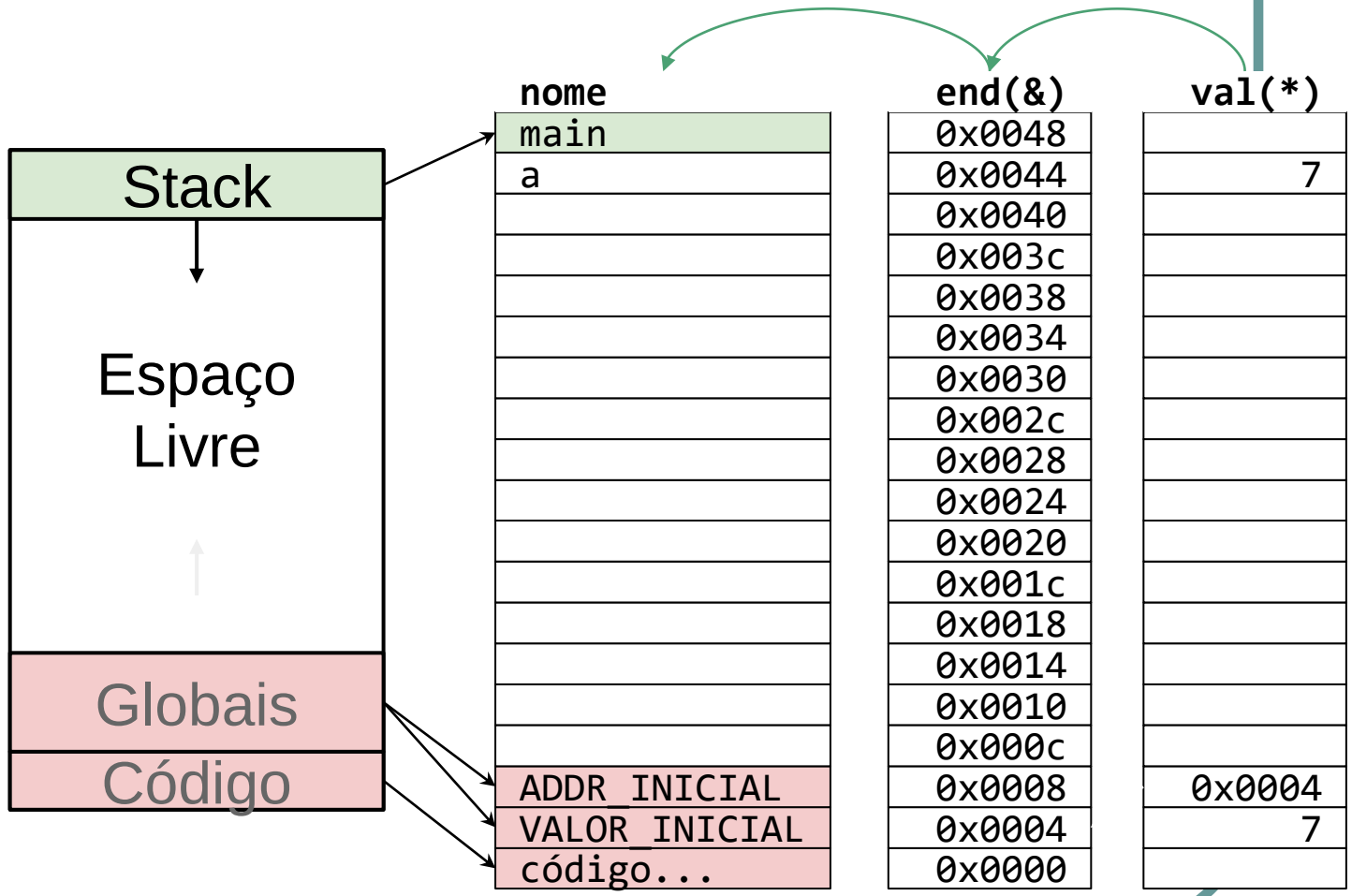
Exemplo

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



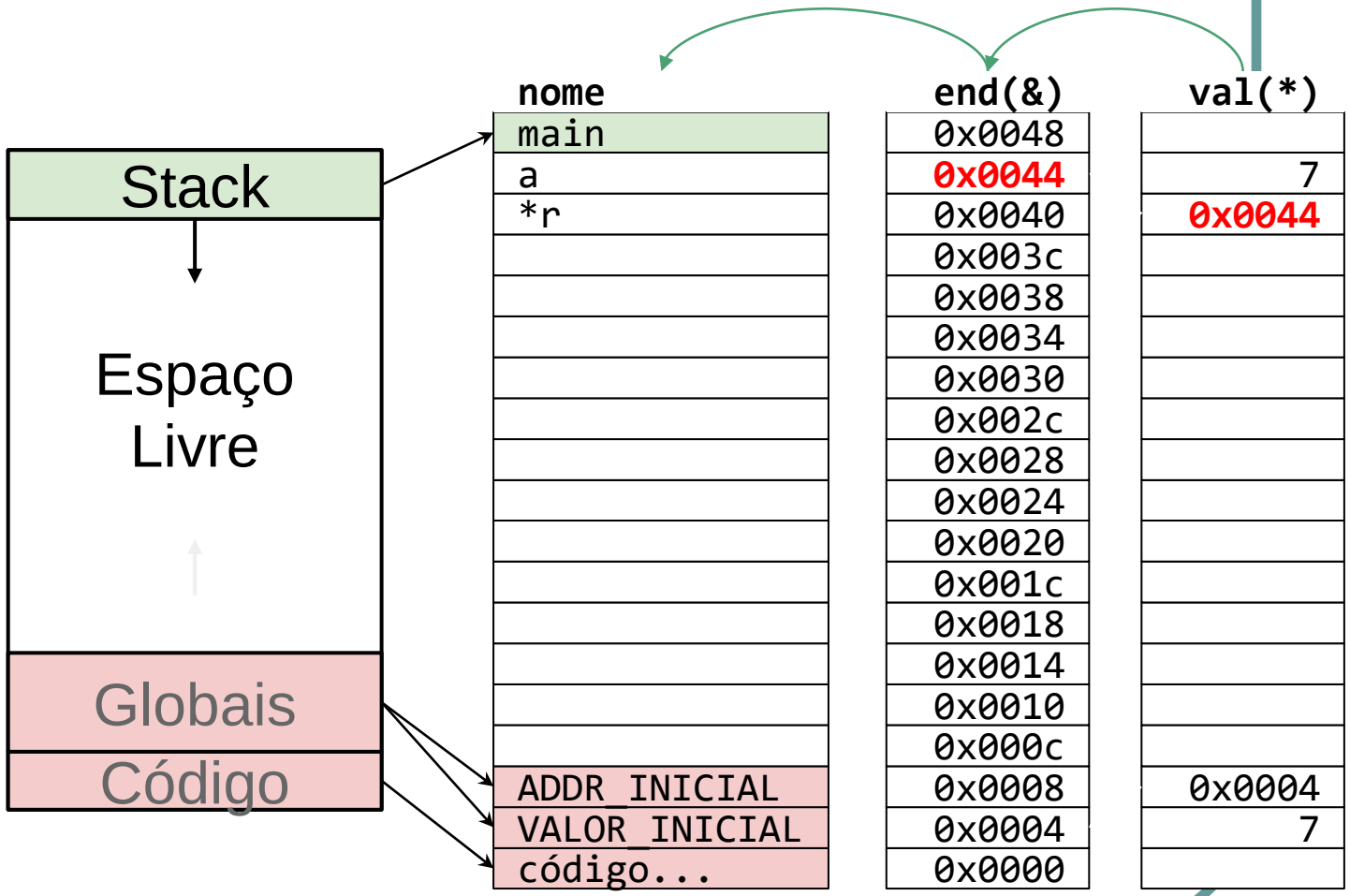
Exemplo

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



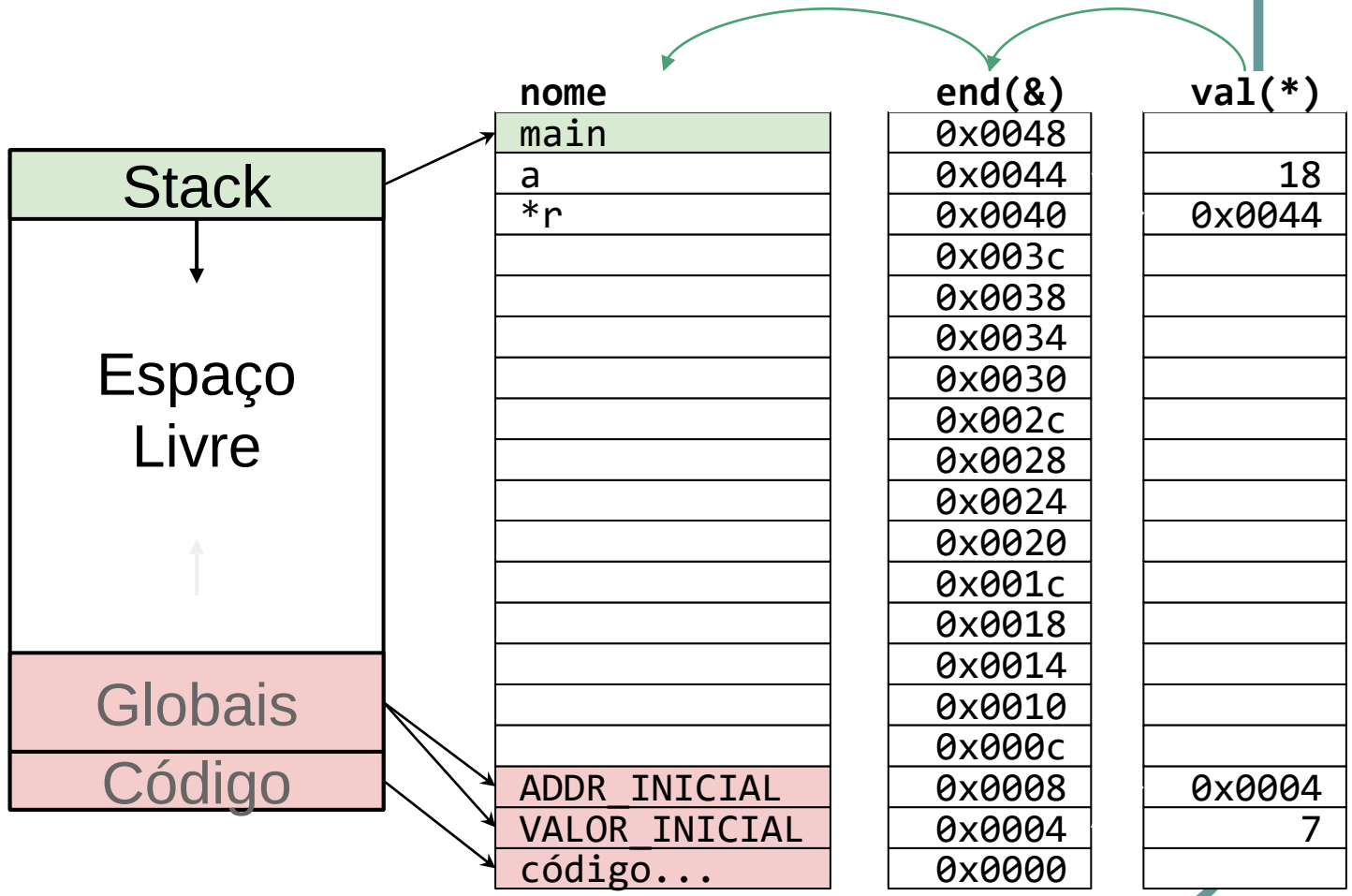
Exemplo

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



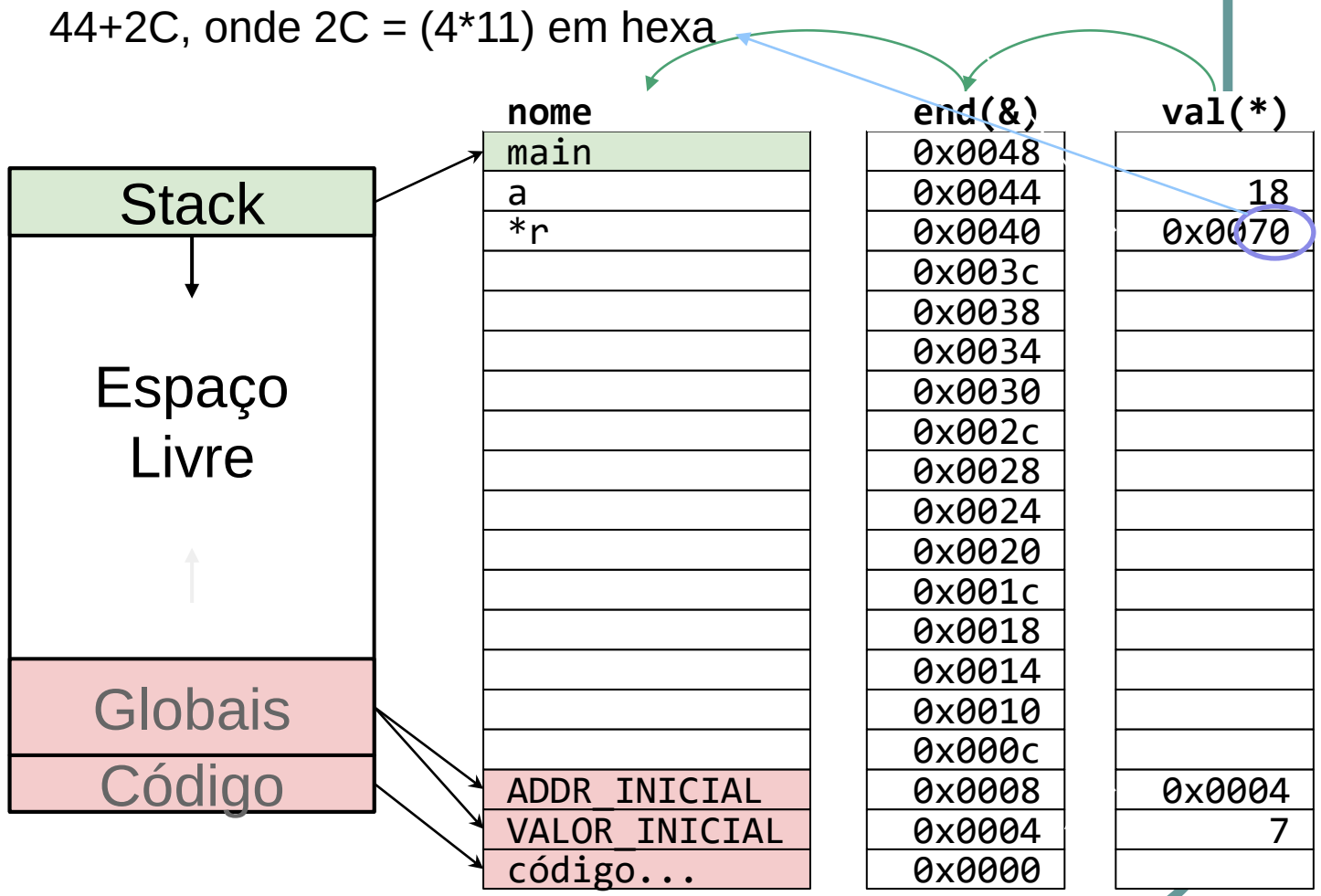
Exempl

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



e agora?!

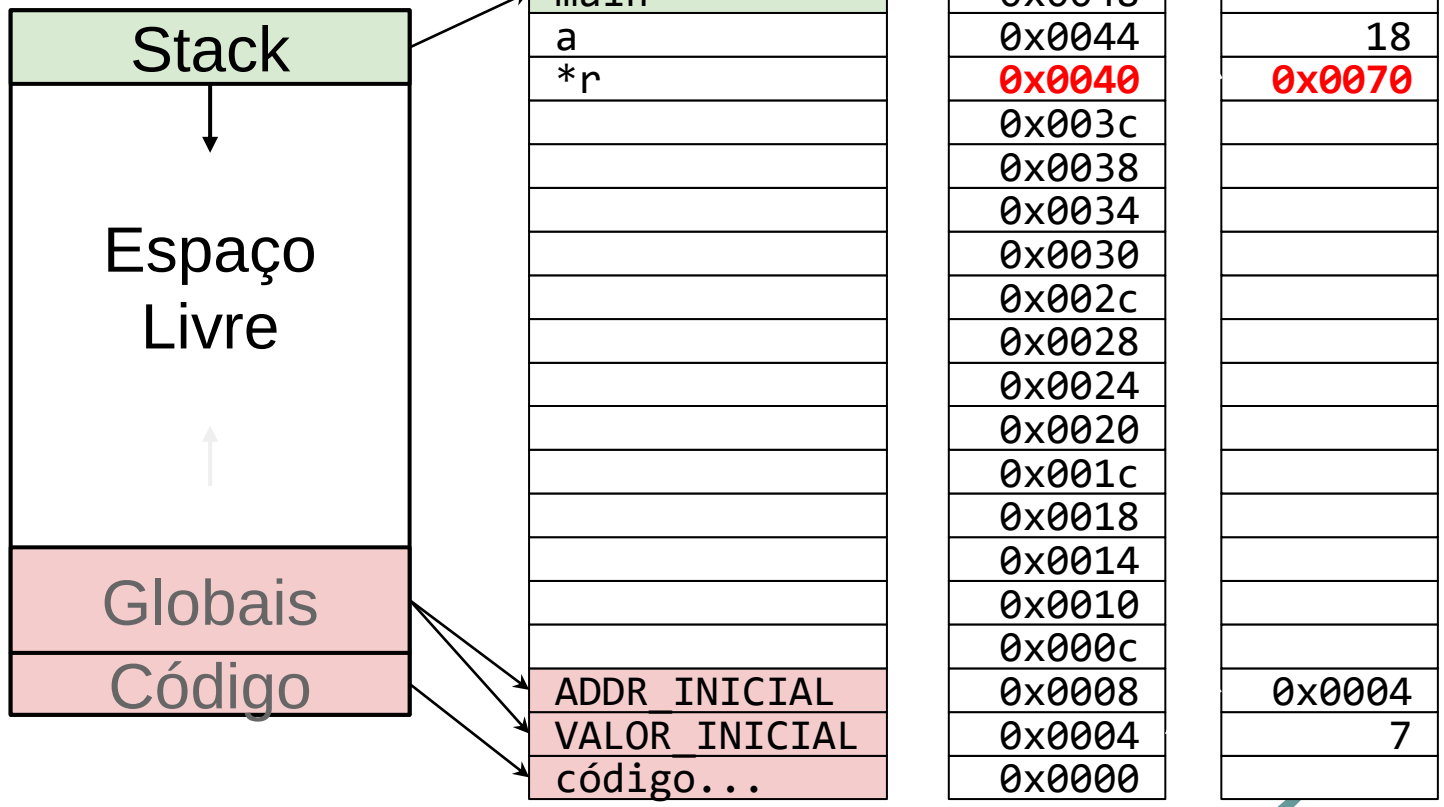
```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



bem esquisito!

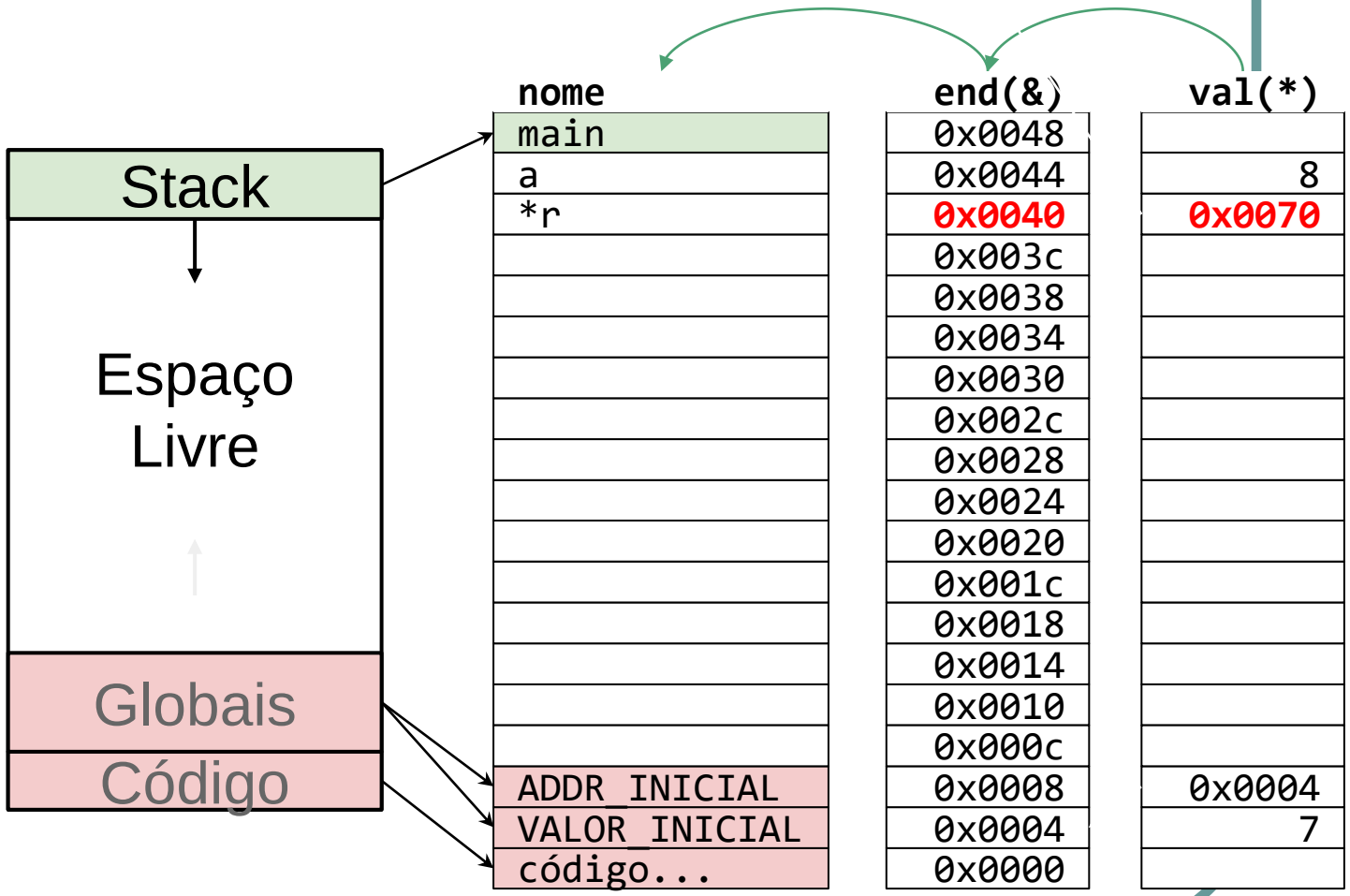
por motivos similares que o programa quebra

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



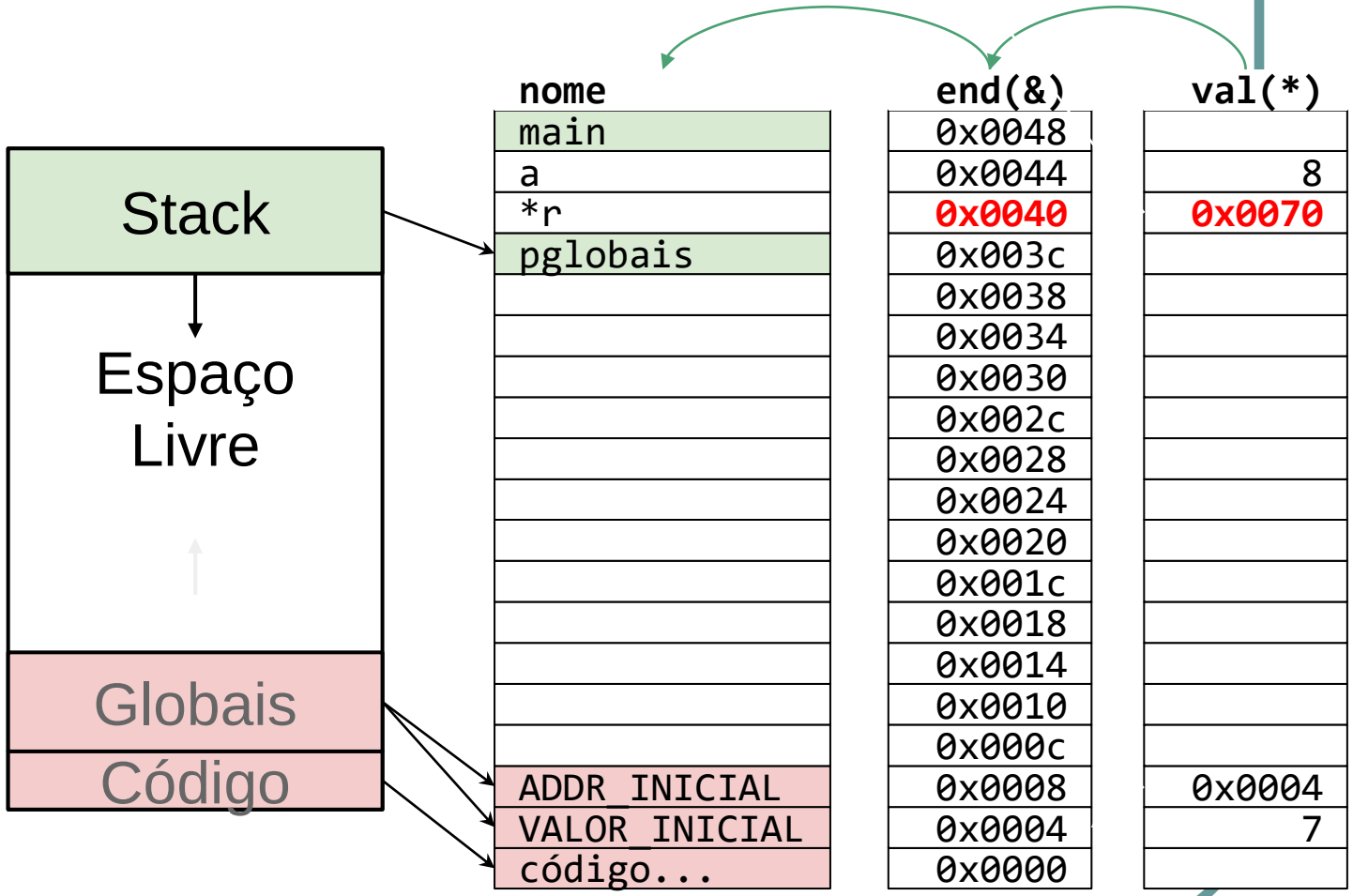
chamando uma função

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



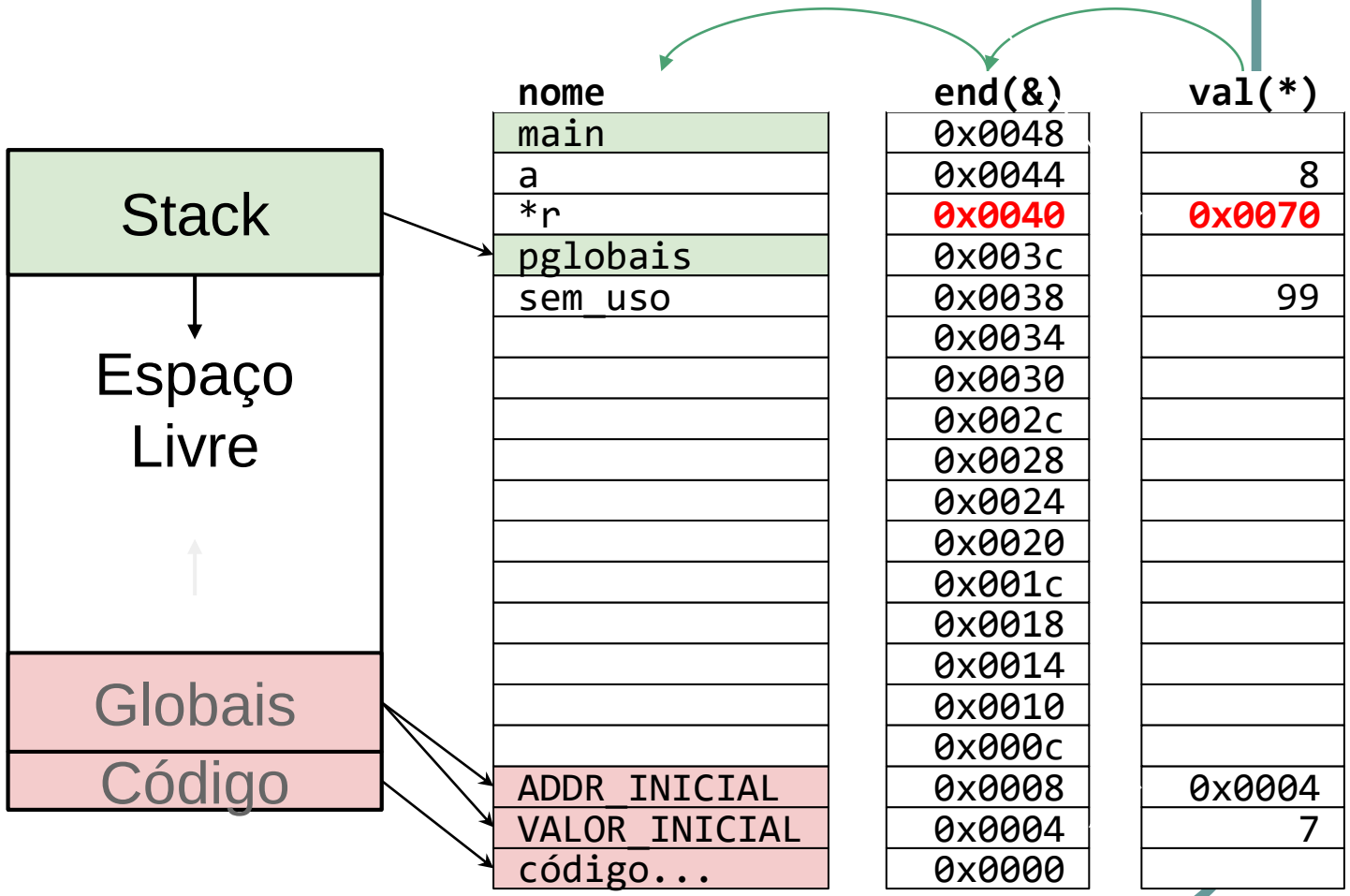
a pilha cresce e temos espaço para a mesma

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



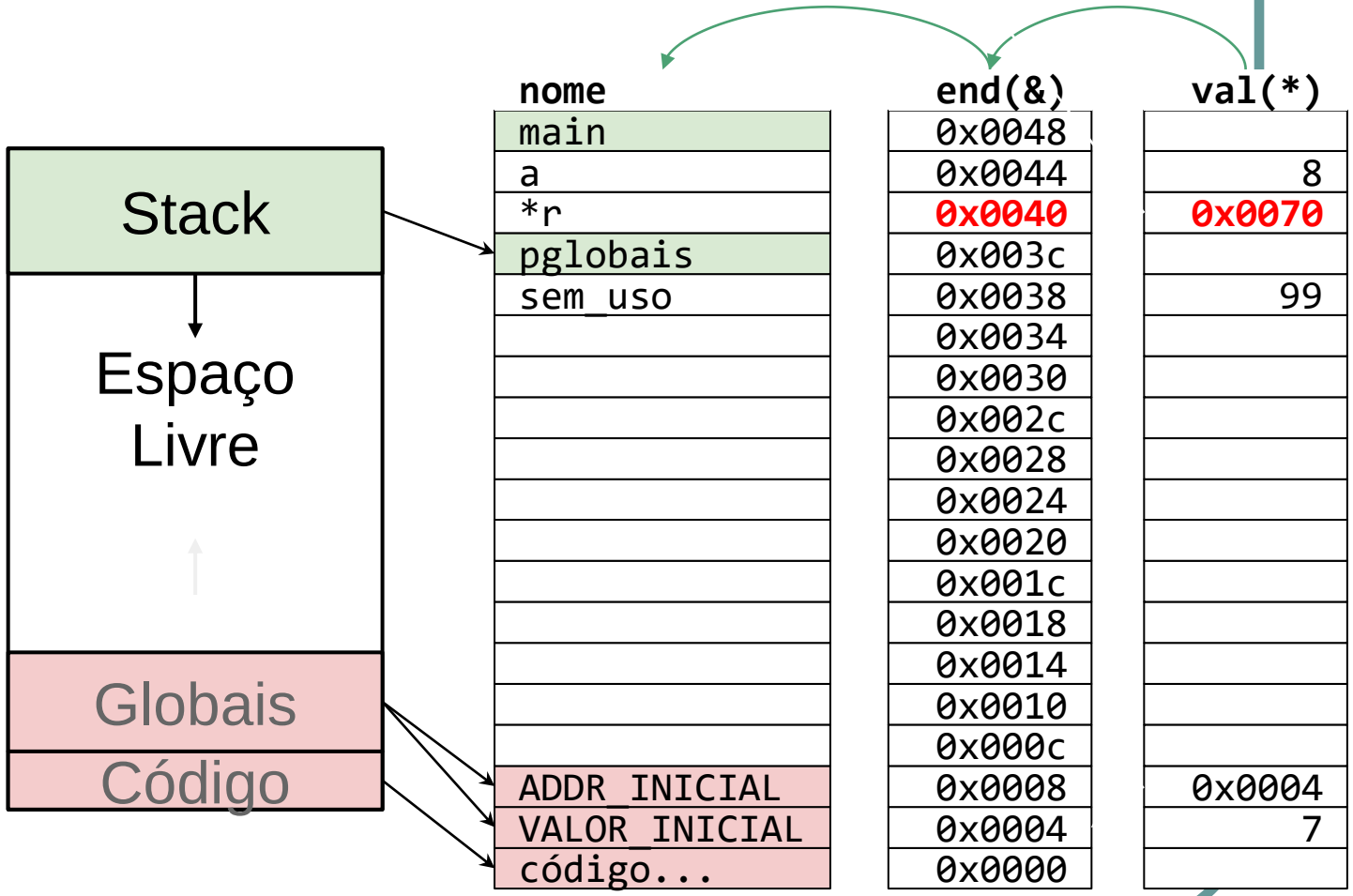
a pilha cresce e temos espaço para a mesma

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



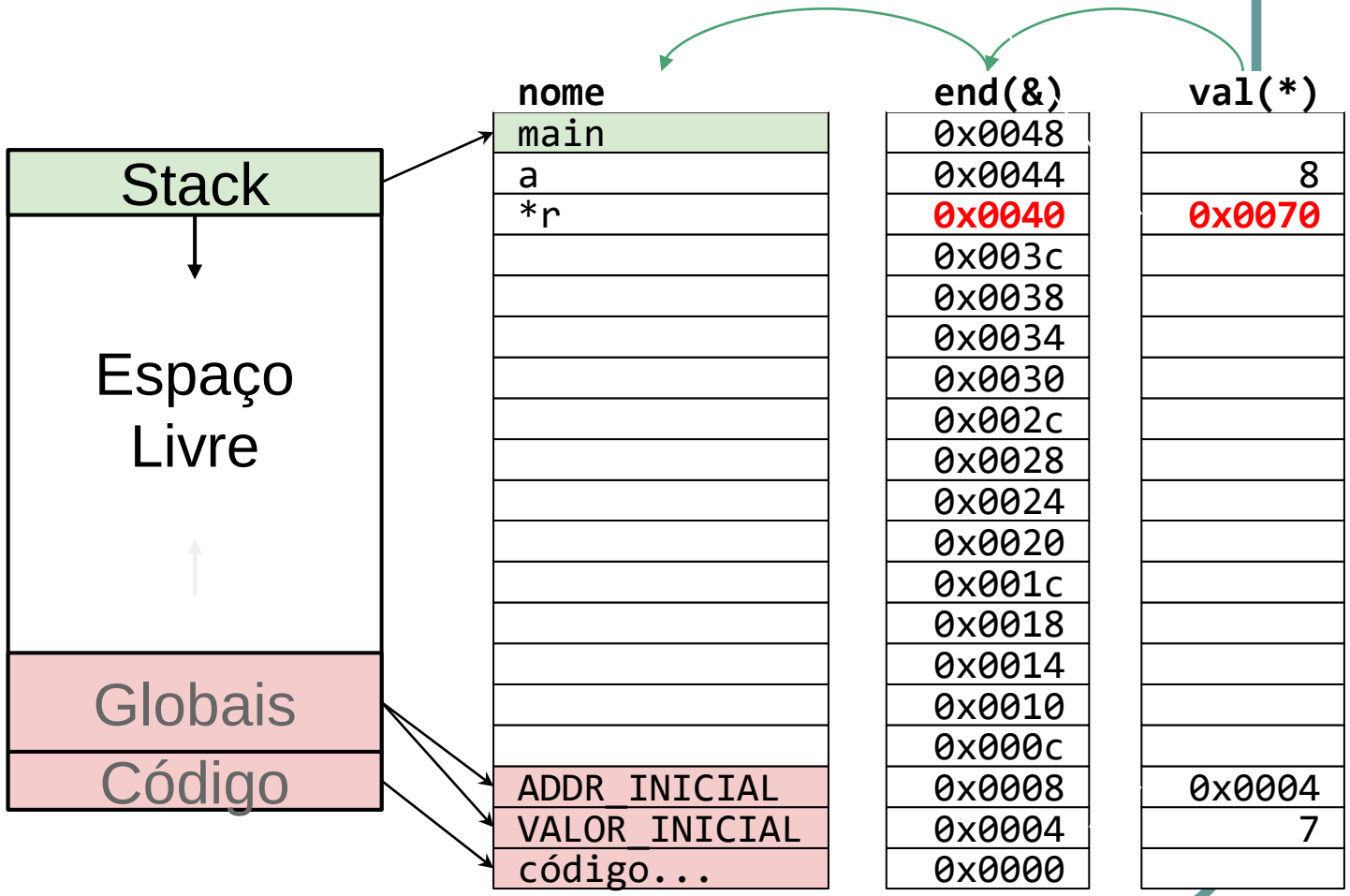
no fim volta a norm

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



no fim volta a norm

```c
#include <stdio.h>

const int VAL_GLB = 7;
const int *END_GLB = &VAL_GLB;

void pglobais(void) {
   int sem_uso = 99;
   printf("end glb %p\n", END_GLB);
   printf("val glb %d\n", VAL_GLB);
}

int main(void) {
  int a = VAL_GLB;
  int *r = &a;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  a+=11;
  r+=11;
  printf("end loc %p\n", r);
  printf("val loc %d\n", a);
  pglobais();
  return 0;
}
```



Stack/Pilha

Então sabemos o que é a pilha. Onde as "funções crescem"

No esquema abaixo desenhamos a pilha crescendo para baixo

O oposto também é comum (depende do livro/arquitetura)

Ponteiro para Struct

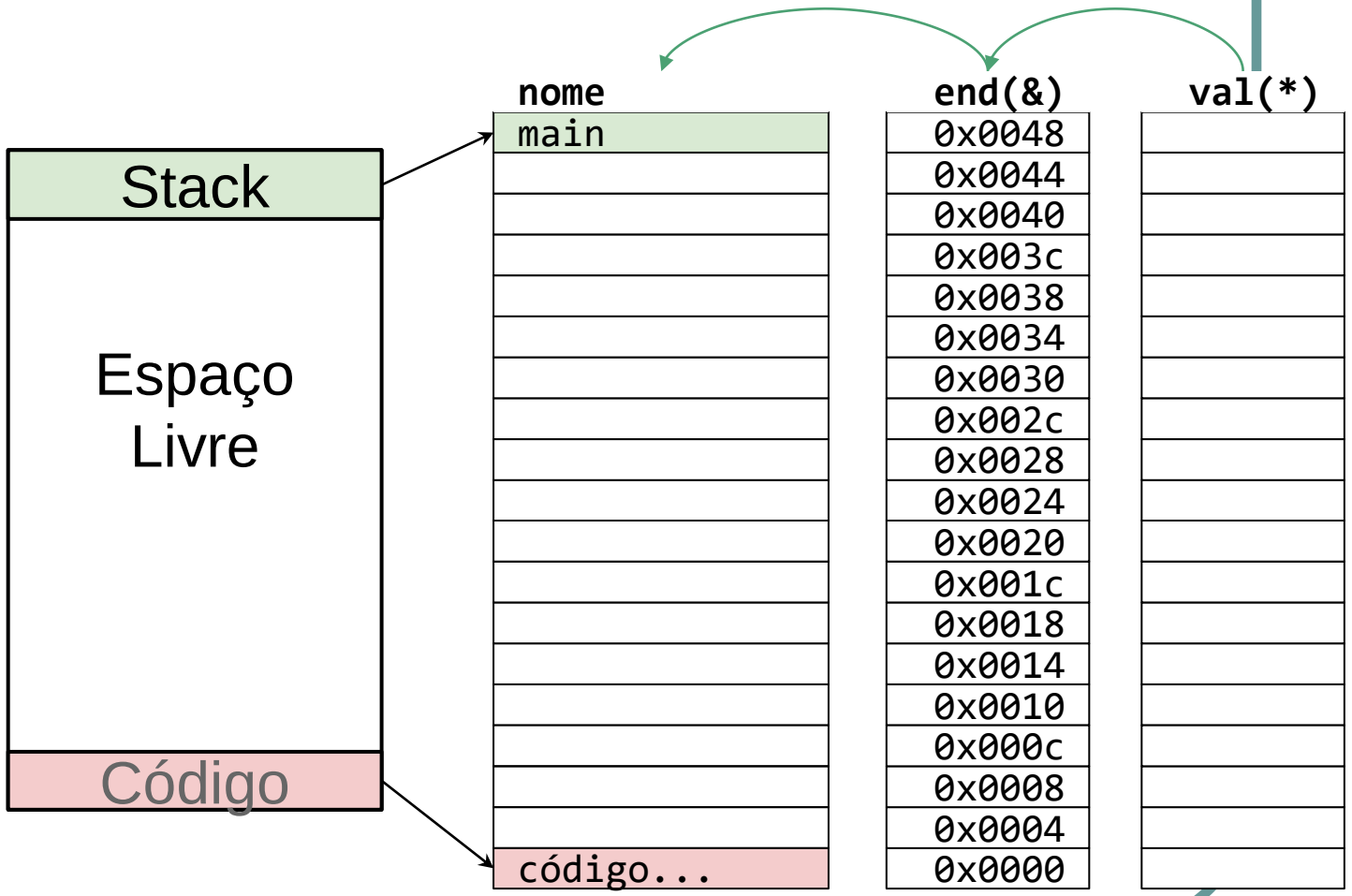
```c
typedef struct {
  double x;
  double y;
} ponto_t;

int main(void) {
  ponto_t ponto;
  ponto_t *ponto_ptr = &ponto;
  ponto_ptr->x = 20;  // equivalente a ponto.x. -> é um "atalho"
  ponto_ptr->y = 99;  // equivalente a ponto.y. -> é um "atalho"
  return 0;
}
```

Função para Iniciar um ponto...

```c
typedef struct {
  double x;
  double y;
} ponto_t;

void inicia_ponto(ponto_t *p) {
  printf("Digite a coord x: ");
  scanf("%lf", p->x);              // antes era (*p).x
  printf("Digite a coord y: ");
  scanf("%lf", p->y);              // antes era (*p).x
}
```

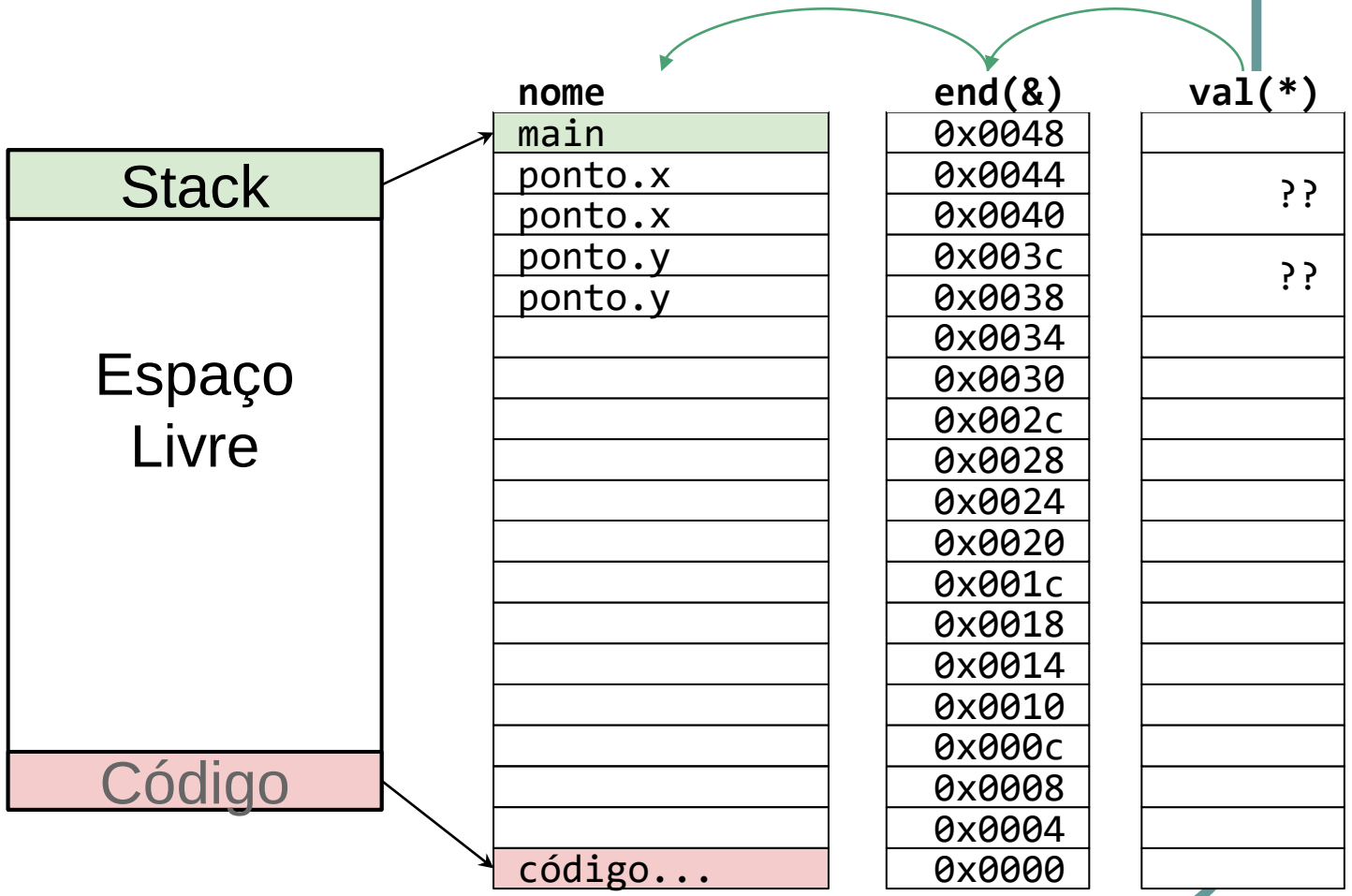
Passagem por Referência (novamente)

```c
typedef struct {
  double x;
  double y;
} ponto_t;

void inicia_ponto(ponto_t *p) {
  printf("Digite a coord x: ");
  scanf("%lf", p->x);
  printf("Digite a coord y: ");
  scanf("%lf", p->y);
}

int main(void) {
  ponto_t ponto;
  inicia_ponto(&ponto);
  return 0;
}
```



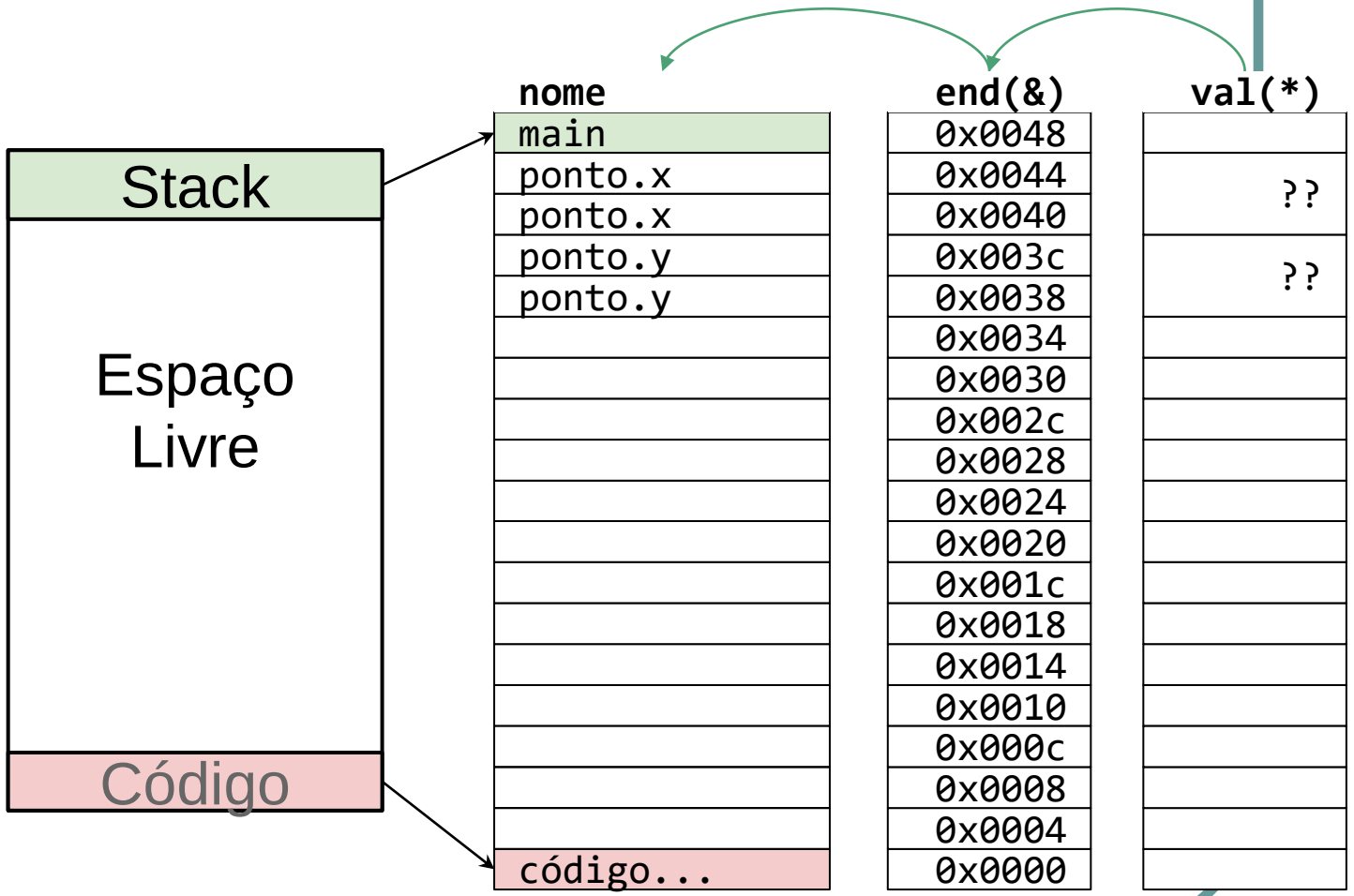
note que o ponto ocupa 4 locais (2 cada double)

```c
typedef struct {
  double x;
  double y;
} ponto_t;

void inicia_ponto(ponto_t *p) {
  printf("Digite a coord x: ");
  scanf("%lf", p->x);
  printf("Digite a coord y: ");
  scanf("%lf", p->y);
}

int main(void) {
  ponto_t ponto;
  inicia_ponto(&ponto);
  return 0;
}
```



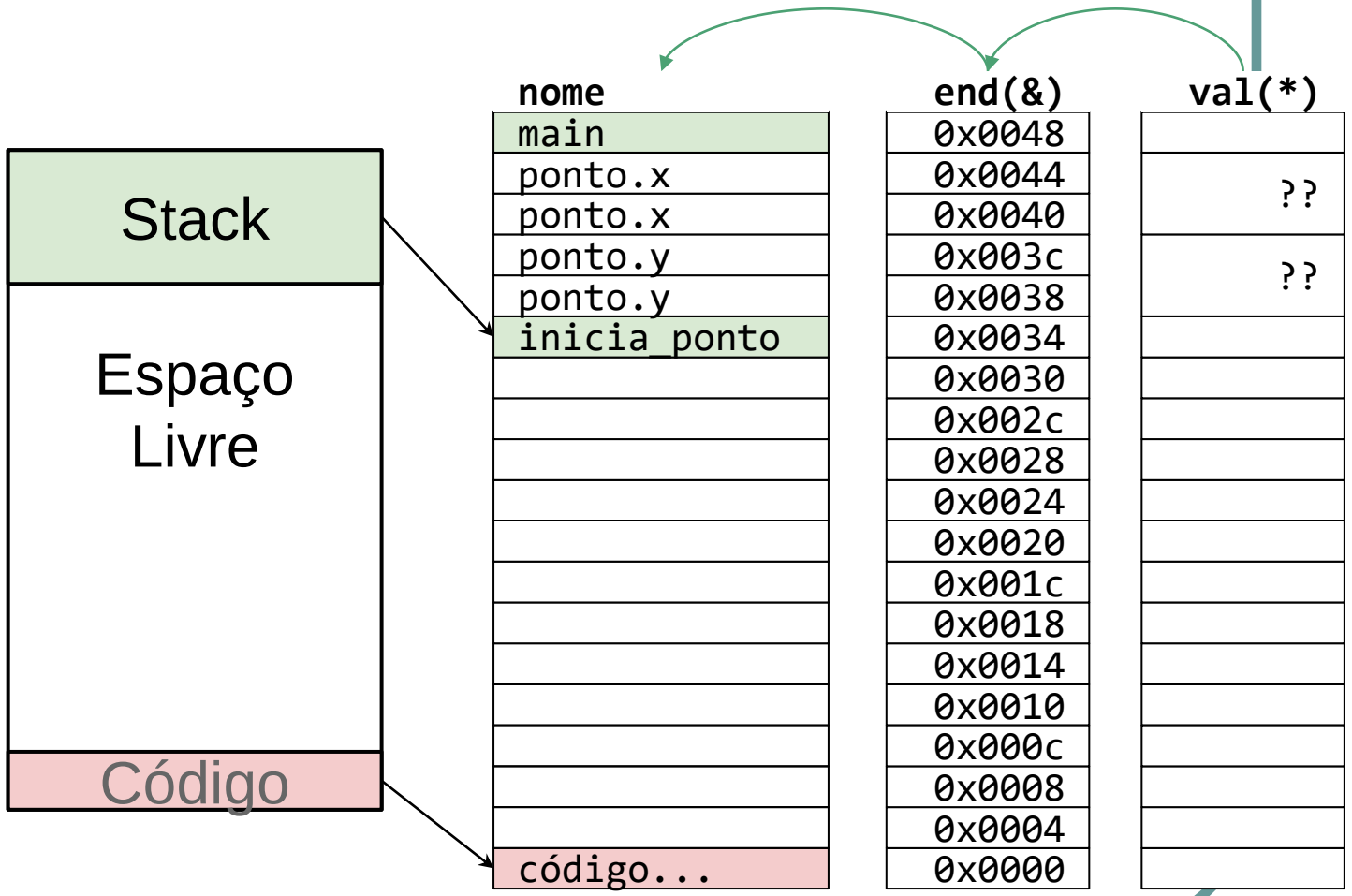
chamamos a função

```c
typedef struct {
  double x;
  double y;
} ponto_t;

void inicia_ponto(ponto_t *p) {
  printf("Digite a coord x: ");
  scanf("%lf", p->x);
  printf("Digite a coord y: ");
  scanf("%lf", p->y);
}

int main(void) {
  ponto_t ponto;
  inicia_ponto(&ponto);
  return 0;
}
```



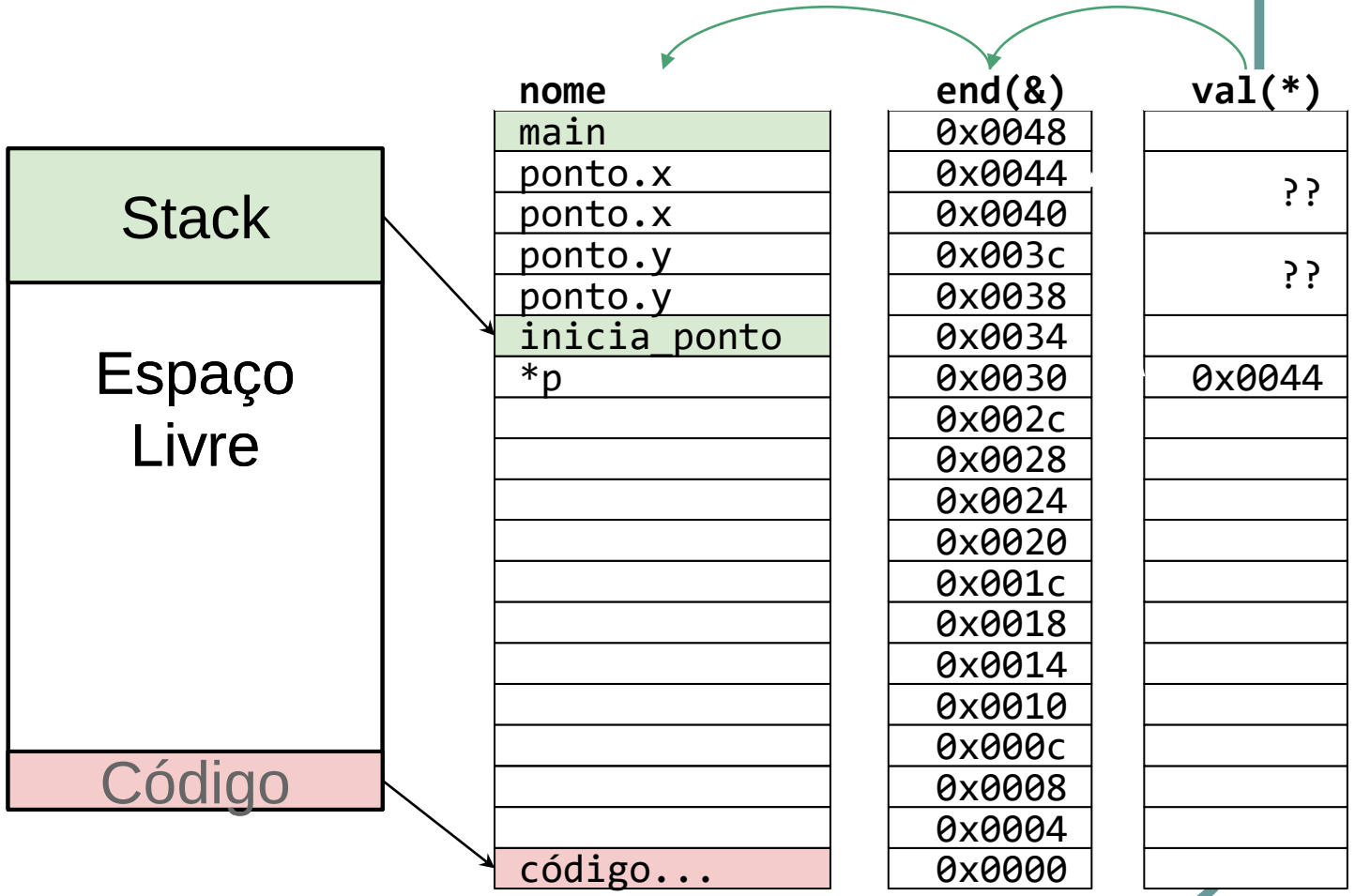
chamamos a função

```c
typedef struct {
  double x;
  double y;
} ponto_t;

void inicia_ponto(ponto_t *p) {
  printf("Digite a coord x: ");
  scanf("%lf", p->x);
  printf("Digite a coord y: ");
  scanf("%lf", p->y);
}

int main(void) {
  ponto_t ponto;
  inicia_ponto(&ponto);
  return 0;
}
```



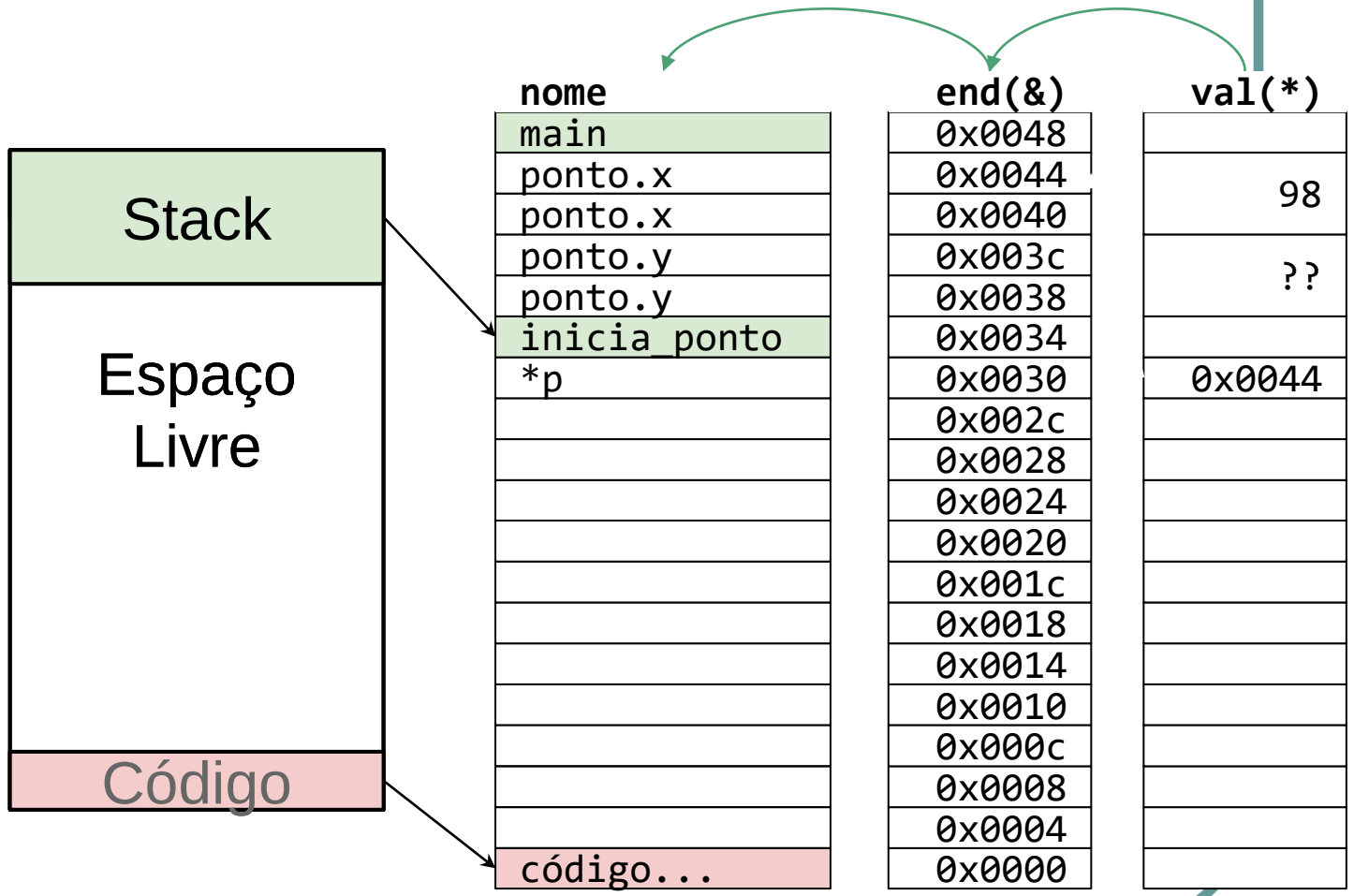
recebeu um ponteiro para ponto

```c
typedef struct {
  double x;
  double y;
} ponto_t;

void inicia_ponto(ponto_t *p) {
  printf("Digite a coord x: ");
  scanf("%lf", p->x);
  printf("Digite a coord y: ");
  scanf("%lf", p->y);
}

int main(void) {
  ponto_t ponto;
  inicia_ponto(&ponto);
  return 0;
}
```



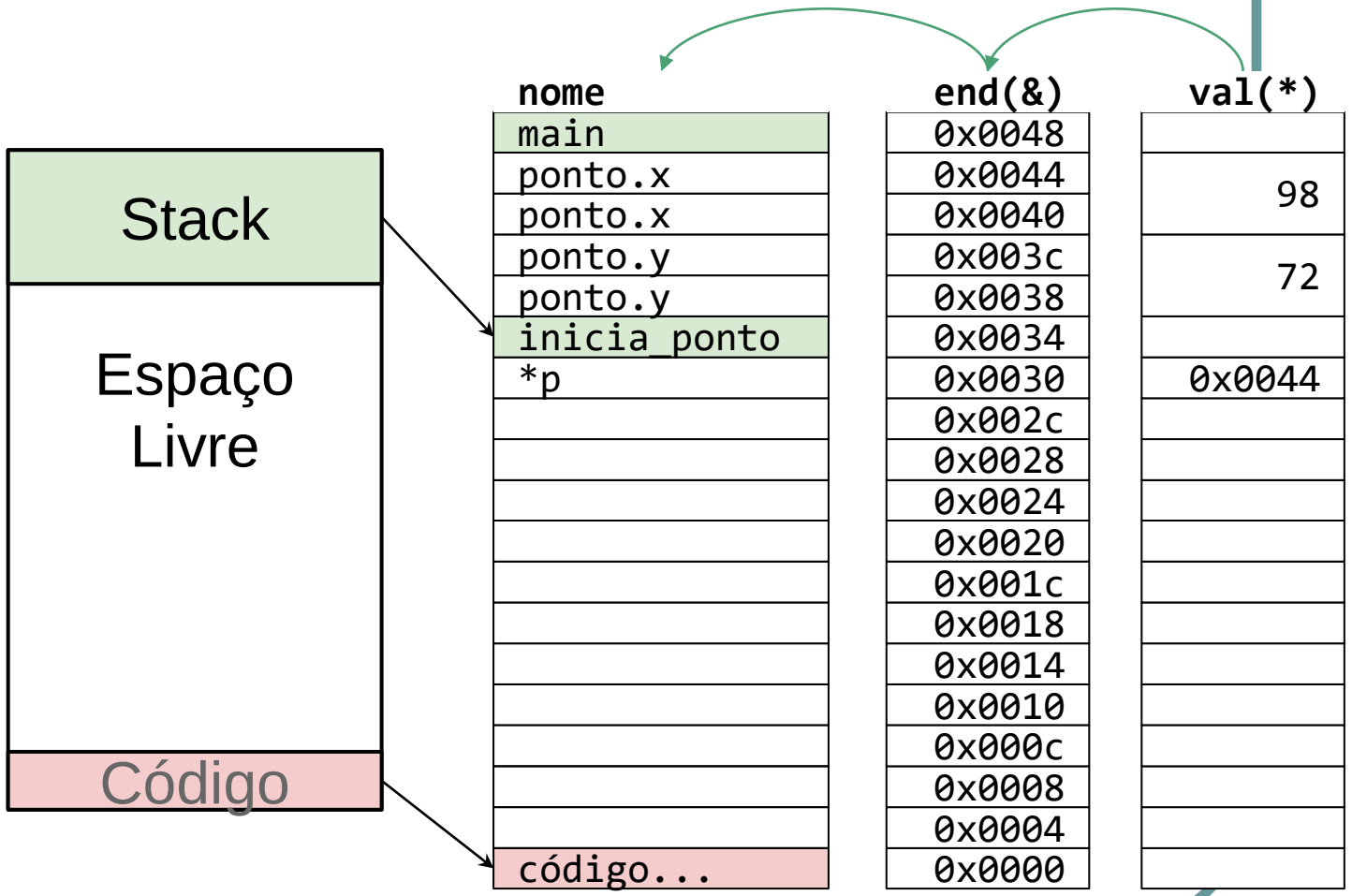
ao ler um valor atualizamos o local correto

```c
typedef struct {
  double x;
  double y;
} ponto_t;

void inicia_ponto(ponto_t *p) {
  printf("Digite a coord x: ");
  scanf("%lf", p->x);
  printf("Digite a coord y: ");
  scanf("%lf", p->y);
}

int main(void) {
  ponto_t ponto;
  inicia_ponto(&ponto);
  return 0;
}
```



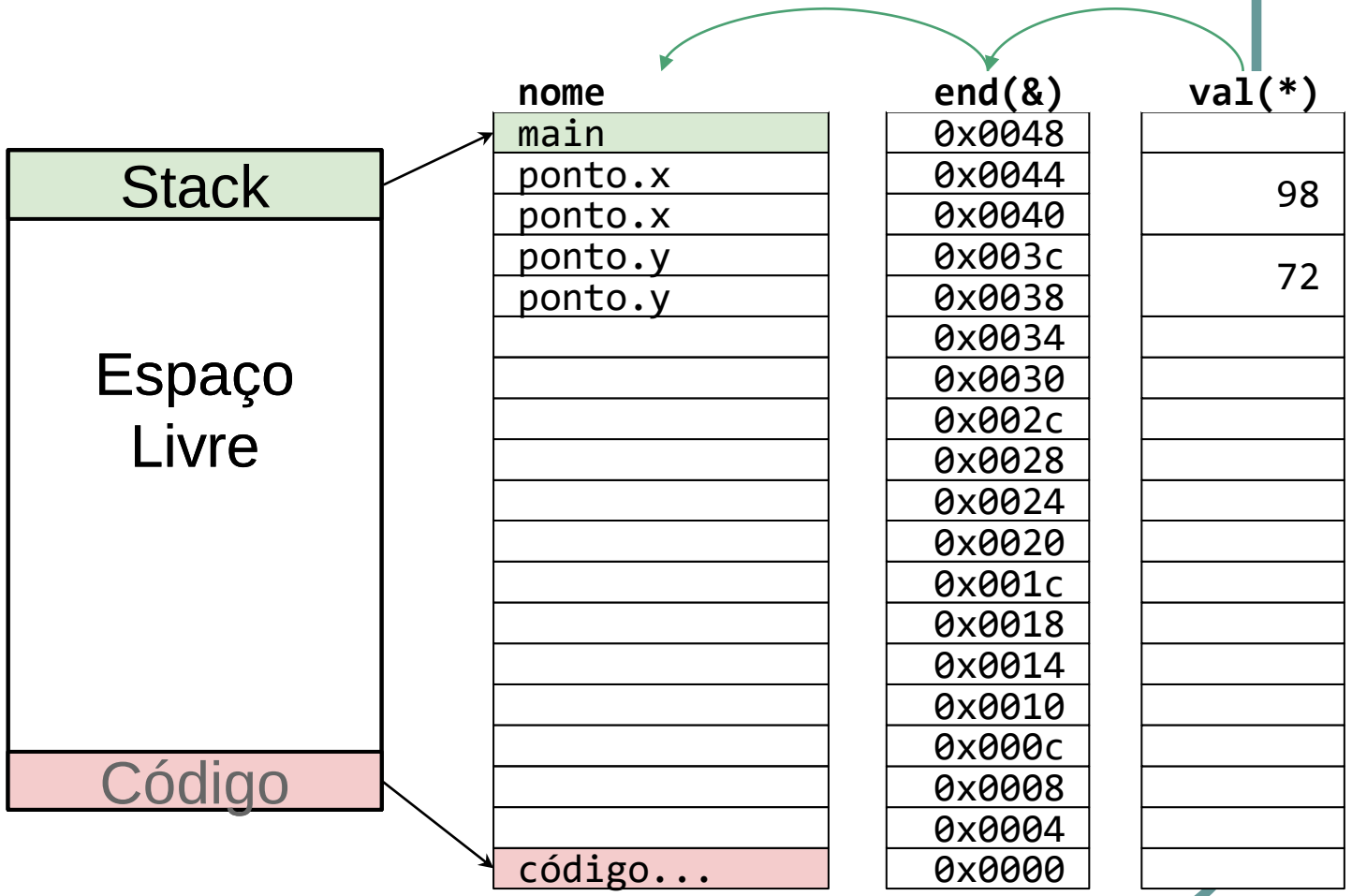
lendo o outro valor

```
typedef struct {
  double x;
  double y;
} ponto_t;

void inicia_ponto(ponto_t *p) {
  printf("Digite a coord x: ");
  scanf("%lf", p->x);
  printf("Digite a coord y: ");
  scanf("%lf", p->y);
}

int main(void) {
  ponto_t ponto;
  inicia_ponto(&ponto);
  return 0;
}
```



ao retornar a função sai da pilha

```
typedef struct {
  double x;
  double y;
} ponto_t;

void inicia_ponto(ponto_t *p) {
  printf("Digite a coord x: ");
  scanf("%lf", p->x);
  printf("Digite a coord y: ");
  scanf("%lf", p->y);
}

int main(void) {
  ponto_t ponto;
  inicia_ponto(&ponto);
  return 0;
}
```



Ponteiros e passagem de parâmetros

Passagem de Parâmetros

- Na linguagem C, os parâmetros de uma função são sempre passados por ***valor***, ou seja, uma cópia do valor do parâmetro é feita e passada para a função.
- Mesmo que esse valor mude dentro da função, nada acontece com o valor de fora da função.

Passagem por valor

```
01  include <stdio.h>
02  include <stdlib.h>
03
04  void soma _ mais _ um(int n){
05    n = n + 1;
06    printf("Dentro da funcao: x = %d\n",n);
07  }
08
09  int main(){
10    int x = 5;
11    printf("Antes da funcao: x = %d\n",x);
12    soma _ mais _ um(x);
13    printf("Depois da funcao: x = %d\n",x);
14    system("pause");
15    return 0;
16  }
```

Saída

```
Antes da funcao: x = 5
Dentro da funcao: x = 6
Depois da funcao: x = 5
```

Passagem por referência

- Quando se quer que o valor da variável mude dentro da função, usa-se passagem de parâmetros por ***referência***.
- Neste tipo de chamada, não se passa para a função o valor da variável, mas a sua ***referência*** (seu endereço na memória);

Passagem por referência

- Utilizando o endereço da variável, qualquer alteração que a variável sofra dentro da função será refletida fora da função. Ex: função scanf()
  - sempre que desejamos ler algo do teclado, passamos para a função **scanf()** o nome da variável onde o dado será armazenado. Essa variável tem seu valor modificado dentro da função **scanf()**, e seu valor pode ser acessado no programa principal

Passagem por referência

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04    int x = 5;
05    printf("Antes do scanf: x = %d\n",x);
06    printf("Digite um numero: ");
07    scanf("%d",&x);
08    printf("Depois do scanf: x = %d\n",x);
09    system("pause");
10    return 0;
11  }
```

Passagem por referência

- Para passar um parâmetro por referência, passamos o ponteiro como parâmetro:

  float sqr (float *num);

- Ao se chamar a função, é necessário agora utilizar o operador “&”, igual como é feito com a função **scanf()**:

  y = sqr(&x);

Passagem por referência

- No corpo da função, é necessário trabalhar corretamente com o ponteiro

Por valor

```
void soma_mais_um(int n){
  n = n + 1;
}
```

Por referência

```
void soma_mais_um(int *n){
  *n = *n + 1;
}
```

Passagem por referência

```
01      include <stdio.h>
02      include <stdlib.h>
03
04      void soma _ mais _ um(int *n){
05        *n = *n + 1;
06         printf("Dentro da funcao: x = %d\n",*n);
07      }
08
09      int main(){
10        int x = 5;
11        printf("Antes da funcao: x = %d\n",x);
12        soma _ mais _ um(&x);
13        printf("Depois da funcao: x = %d\n",x);
14        system("pause");
15        return 0;
16      }
```

```
Saída    Antes da funcao: x = 5
         Dentro da funcao: x = 6
         Depois da funcao: x = 6
```

Exercício

- Crie uma função que troque o valor de dois números inteiros passados por referência.

Exercício

```
void Troca (int*a,int*b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Arrays como parâmetros

- Para utilizar arrays como parâmetros de funções alguns cuidados simples são necessários.

Arrays como parâmetros

- Arrays são sempre passados por referência para uma função;
  - A passagem de arrays ***por referência*** evita a cópia desnecessária de grandes quantidades de dados para outras áreas de memória durante a chamada da função, o que afetaria o desempenho do programa.

Arrays como parâmetros

- É necessário declarar um segundo parâmetro (em geral uma variável inteira) para passar para a função o tamanho do array separadamente.
  - Quando passamos um array por parâmetro, independente do seu tipo, o que é de fato passado é o endereço do primeiro elemento do array.

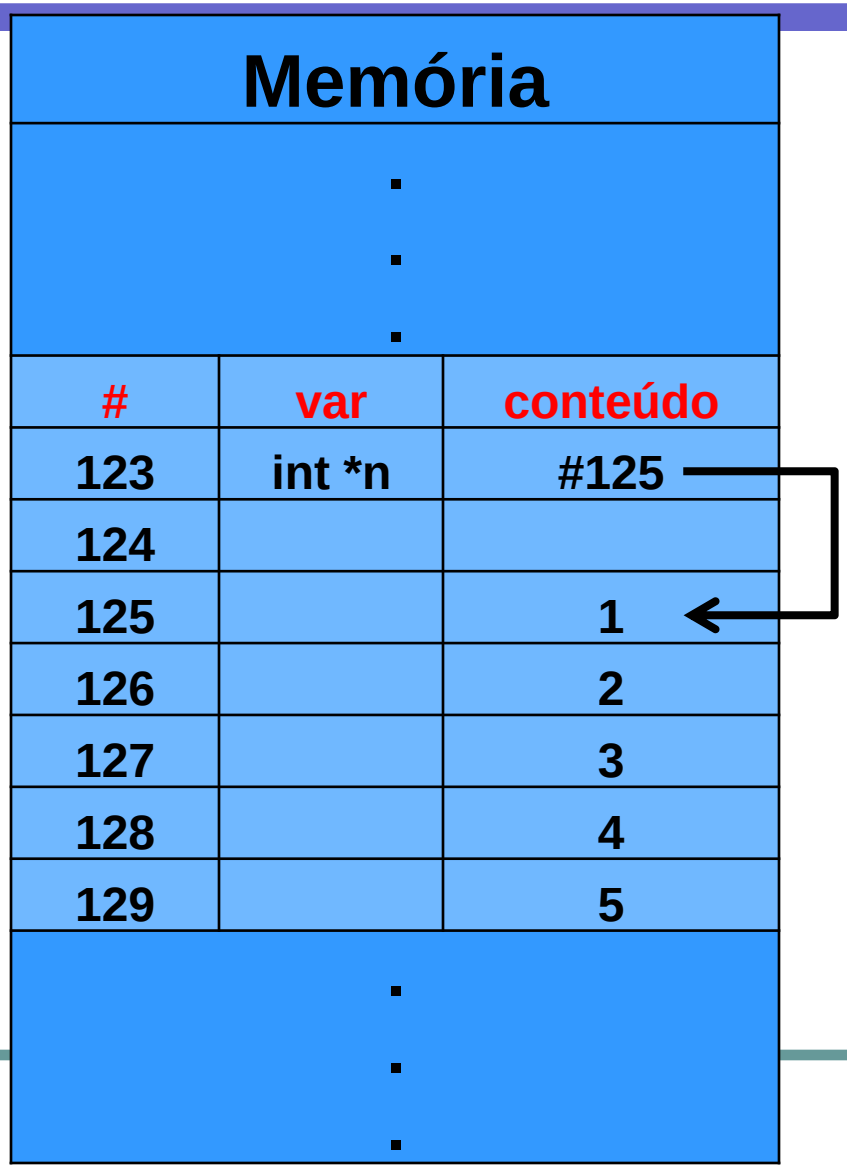
Arrays como parâmetros

- Na passagem de um array como parâmetro de uma função podemos declarar a função de diferentes maneiras, todas equivalentes:

```
void imprime (int *m, int n);
void imprime (int m[], int n);
void imprime (int m[5], int n);
```

Arrays como parâmetros

```
void imprime (int *m,int n){
int i;
for (i=0; i< n;i++)
    printf ("%d \n", m[i]);
}

int main (){
    int n[5] = {1,2,3,4,5};
    imprime(n,5);
    return 0;
}
```



Arrays como parâmetros

- Vimos que para arrays, não é necessário especificar o número de elementos para a função.

  void imprime (int*m, int n);

  void imprime (int m[], int n);

- No entanto, para arrays com mais de uma dimensão, é necessário especificar o tamanho de todas as dimensões, exceto a primeira

  void imprime (int m[][5], int n);

Arrays como parâmetros

- Na passagem de um array para uma função, o compilador precisar saber o tamanho de cada elemento, não o número de elementos.
- Uma matriz pode ser interpretada como um array de arrays.
  - **int m[4][5]**: array de 4 elementos onde cada elemento é um array de 5 posições inteiras.

Arrays como parâmetros

- Logo, o compilador precisa saber o tamanho de cada elemento do array.

```
int m[4][5]
void imprime (int m[][5], int n);
```

- Na notação acima, informamos ao compilador que estamos passando um array, onde cada elemento dele é outro array de 5 posições inteiras.

Arrays como parâmetros

- Isso é necessário para que o programa saiba que o array possui mais de uma dimensão e mantenha a notação de um conjunto de colchetes por dimensão.

- As notações abaixo funcionam para arrays com mais de uma dimensão. Mas o array é tratado como se tivesse apenas uma dimensão dentro da função

```
void imprime (int*m, int n);
void imprime (int m[], int n);
```

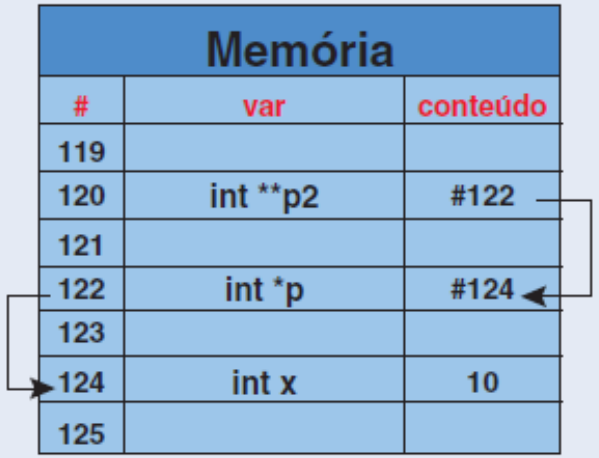
Ponteiro para ponteiro

- Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo.
- Podemos declarar um ponteiro para um ponteiro com a seguinte notação
  - tipo_ponteiro **nome_ponteiro;
  - **nome_ponteiro é o conteúdo final da variável apontada;
  - *nome_ponteiro é o conteúdo do ponteiro intermediário.

Ponteiro para ponteiro

Exemplo: ponteiro para ponteiro

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04    int x = 10;
05    int *p = &x;
06    int **p2 = &p;
07    //Endereço em p2
08    printf("Endereco em p2: %p\n",p2);
09    //Conteudo do endereço
10    printf("Conteudo em *p2: %p\n",*p2);
11    //Conteudo do endereço do endereço
12    printf("Conteudo em **p2: %d\n",**p2);
13    system("pause");
14    return 0;
15  }
```



Ponteiro para ponteiro

- É a quantidade de asteriscos (*) na declaração do ponteiro que indica o número de níveis de apontamento que ele possui.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04    //variável inteira
05    int x;
06    //ponteiro para um inteiro (1 nível)
07    int *p1;
08    //ponteiro para ponteiro de inteiro (2 níveis)
09    int **p2;
10    //ponteiro para ponteiro para ponteiro de inteiro(3 níveis)
11    int ***p3;
12    system("pause");
13    return 0;
14  }
```

Ponteiro para ponteiro

- Ex.:

```
char letra='a';
char *ptrChar;
char **ptrPtrChar;
char ***ptrPtr;
ptrChar = &letra;
ptrPtrChar = &ptrChar;
ptrPtr = &ptrPtrChar;
```