Apontadores e Variáveis Dinâmicas

Prof. Alberto Costa Neto

O problema

- **Dificuldade em definir quantidade exata de dados armazenados:**
  - Como saber quantos alunos teremos em uma Universidade?
  - Ou quantos veículos teremos em um país?
  - Ou quantos computadores estão ligados à Internet?
- Além disso, **parte dos dados pode ser necessária apenas em certas etapas do processamento.**
- Escolher o tamanho das EDs estaticamente normalmente incorre em **desperdício** ou **limita o tamanho da entrada de dados.**

Alocação de Memória

- Até o momento, conhecemos os seguinte tipos de variáveis:
  - **Variáveis Globais**
    - Existem enquanto o programa estiver executando.
  - **Variáveis Locais**
    - Existem enquanto uma função estiver executando
- Porém, nas próximas EDs, utilizaremos **Variáveis Dinâmicas**
  - Alocadas e desalocadas a qualquer momento

Alocação Estática de Memória

- Toda a memória que pode vir a ser necessária é **alocada toda de uma vez**
- **Sem considerar a quantidade realmente necessária** em cada execução do programa
- O **máximo** de alocação possível **é ditado pelo hardware**, ou seja, pelo tamanho da memória “endereçável”

Alocação Estática de Memória (Exemplos)

`int quantidades[1000]`

- Espaço contíguo na memória para 1000 valores do tipo int
- Se cada int ocupa 4 bytes, 4000 bytes

`char nome[100]`

- Espaço contíguo na memória para 100 valores do tipo char
- Se cada char ocupa 1 byte, 100 bytes

Alocação Estática x Dinâmica

Armazenar nome e sobrenome dos alunos:

- Com 3000 espaços de memória disponíveis
- Usando vetor de string (alocado estaticamente)
- 100 caracteres (Tamanho máximo do nome inteiro)

Com isso, podemos então comportar até 30 pessoas

- Não é o ideal pois a maioria dos nomes não requer os 100 caracteres

Na alocação dinâmica não é necessário definir de antemão o tamanho máximo para os nomes.

Alocação Dinâmica de Memória

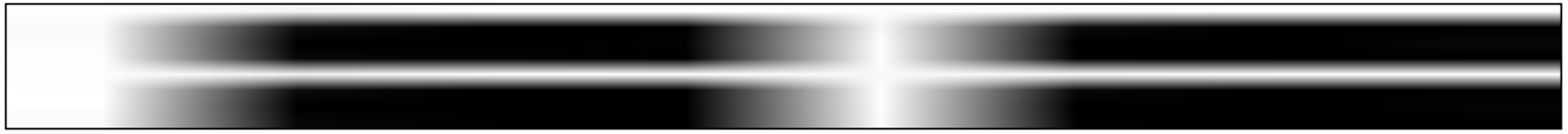
Alocação Dinâmica de Memória

- **Alocação de memória sob demanda**
- As áreas de memória podem ser alocadas, liberadas e realocadas para diferentes propósitos, durante a execução do programa
- Em C usamos **malloc(n)** para alocar um bloco de memória de tamanho **n** bytes.
- É responsabilidade do programador desalocar (liberar) a memória após seu uso

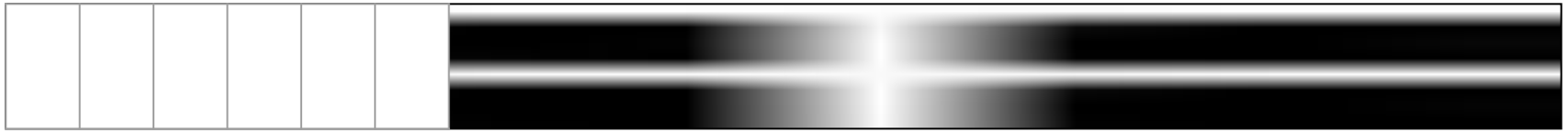
```
void *malloc(size_t size); // size_t == unsigned int
```

Alocação Dinâmica de Memória

- Espaço endereçável (3000) ainda livre:

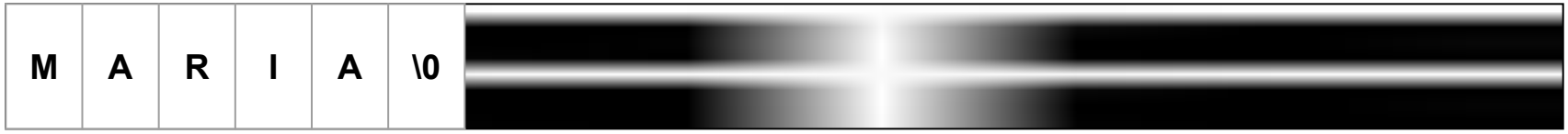


  - Alocado espaço para o nome MARIA
  - 5 bytes para o nome + um byte para o caractere NULL (\0). Total: 6 bytes
    - **malloc (6)**



Alocação Dinâmica de Memória

- Escrevemos MARIA no espaço



- Alocamos e escrevemos PATRICIA

Alocação Dinâmica de Memória

- Endereços não necessariamente contíguos
- Gerenciador de memória do S.O. aloca blocos de memória que estão livres
- Gerenciador de memória controla espaços ocupados e livres
- Inicialmente a memória alocada contém lixo. Por isso, é necessário inicializar
- Em C, liberamos a memória usando free(p)

```
void free(void *ptr);
```

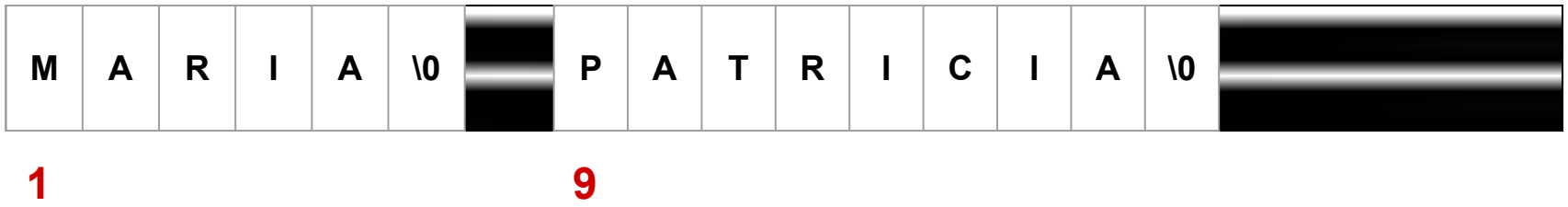
Problemas comuns na Alocação de Memória Dinâmica

- **Liberar memória é responsabilidade do programador**
- Por isso é comum que erros sejam cometidos, como:
  - Acessar memória que não pertence ao programa,
    - Invasão de memória (“memory violation”)
  - Acessar um endereço válido, mas que pertence a outra variável dinâmica e tem outro propósito
- Fragmentação
  - Blocos livres de memória não contíguos

- EDs encadeadas lidam melhor com a fragmentação

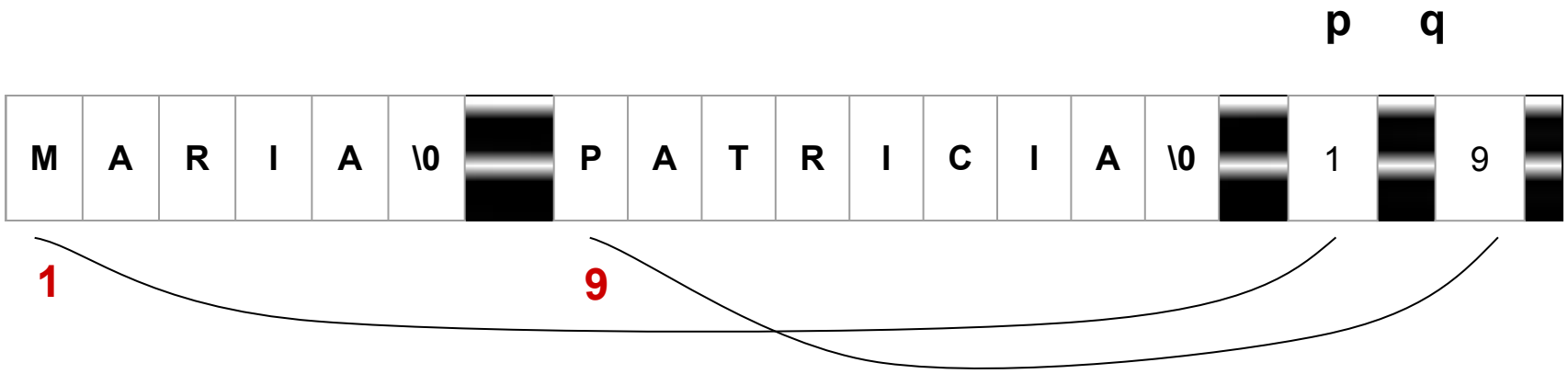
Endereçamento em alocação dinâmica

- Precisamos saber os endereços dos espaços de memória usados



- **Ponteiros** são **variáveis** que armazenam o endereço na própria memória

```
char *p; char *q;
```



Alocação dinâmica em C

- Funções disponíveis em stdlib

void *malloc (size_t n);

- Aloca n bytes e retorna um apontador para esta posição da memória

void *calloc (size_t n, size_t size);

- Aloca (n*size) bytes, atribui zero a todos eles e retorna um apontador para esta posição da memória

void *realloc (void *p, size_t n);

- Aloca uma nova área de memória de tamanho n, copia o conteúdo da memória apontado por p para esta área e retorna o apontador para esta posição da memória.

void free (void *p);

- Libera a memória alocada referenciada por p. Não muda o valor do apontador.

MALLOC

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char** argv)
{
    int *p = NULL;
    int n;
    ... /* Obtem o valor de n de alguma forma */
    p = (int *)malloc(n*sizeof(int));
    if (!p) // equivalente: p==NULL
    {
         printf ("** Erro: Memoria Insuficiente **\n");
         exit(EXIT_FAILURE);
    }
    ...
    if (p!=NULL) {free(p); p=NULL;}
   return EXIT_SUCCESS;
}
```

Alocada memória suficiente para se colocar n números inteiros

CALLOC

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char** argv)
{
    int *p=NULL;
    int n;
    ...
    p = (int *)calloc(n, sizeof(int));
    if (!p) // equivalente: p==NULL
    {
       printf ("** Erro: Memoria Insuficiente **\n");
       exit(EXIT_FAILURE);
    }
    ...
    if (p!=NULL) {free(p);p=NULL;}
    return EXIT_SUCCESS;
}
```

Alocada memória suficiente para se colocar n números inteiros e coloca todos os bits com valor 0.

FREE

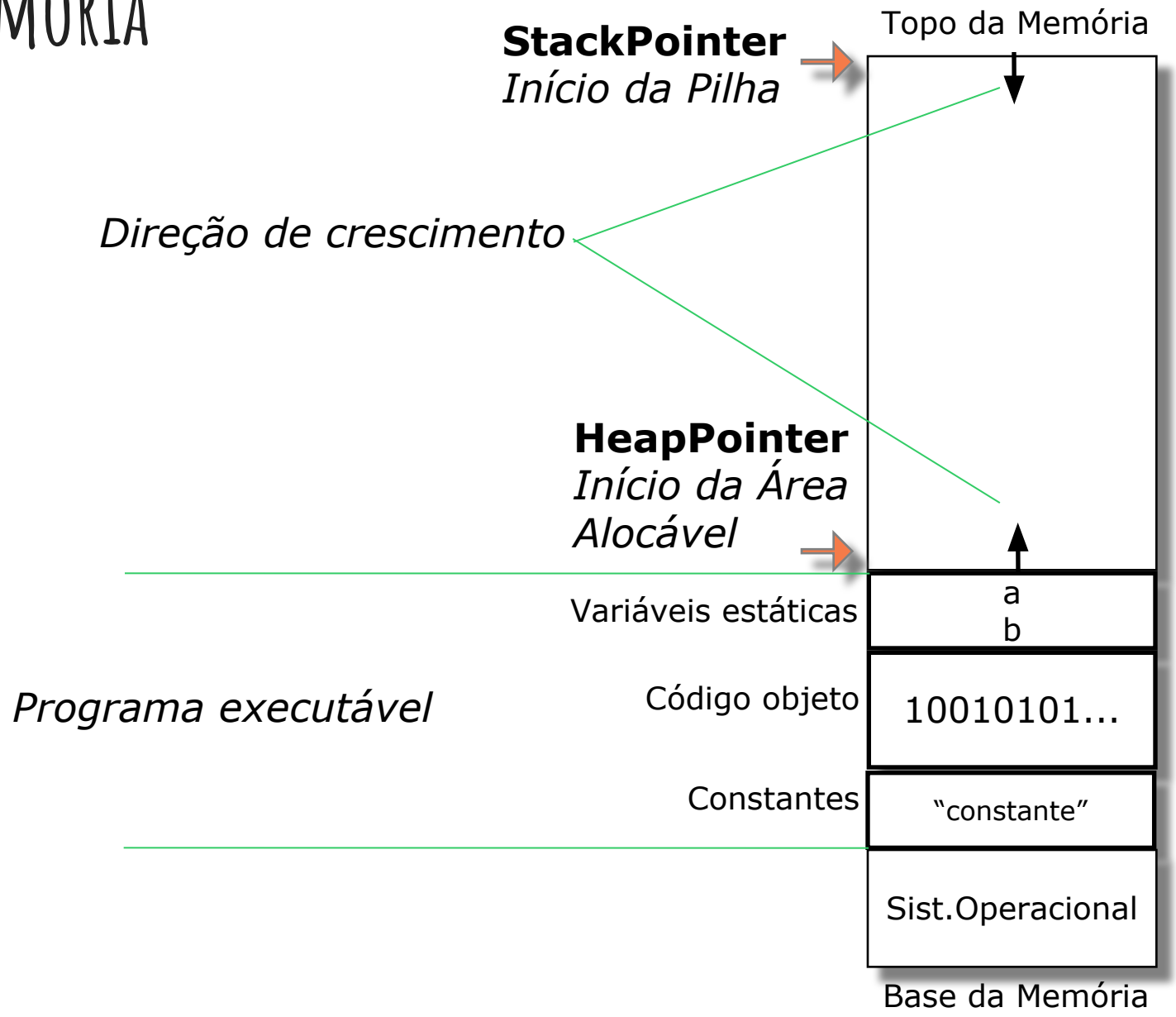
```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char** argv)
{
   int *p = NULL;
   int n;
   ... /* Obtem o valor de n em algum lugar */
   p = (int *)malloc(a*sizeof(int));
   if (!p) // equivalente: p==NULL
   {
      printf ("** Erro: Memoria Insuficiente **\n");
      exit(EXIT_FAILURE);
   }
   ...
   if (p!=NULL) {free(p);p=NULL;}
   return EXIT_SUCCESS;
}
```

Mais detalhes sobre alocação De Memória

Alocação da Memória

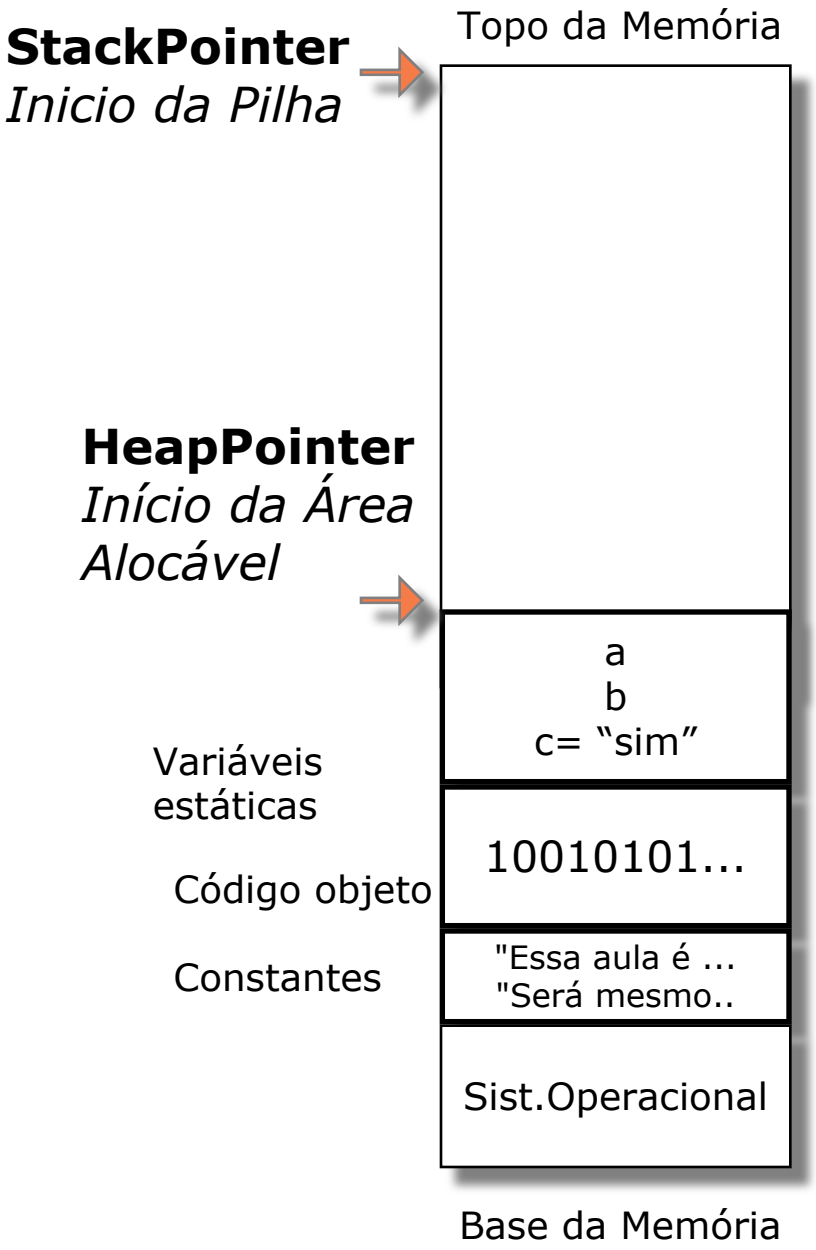
- Constantes: codificadas dentro do código objeto em tempo de compilação
- Variáveis globais (estáticas): alocadas no início da execução do programa
- Variáveis locais (funções ou métodos): alocadas através da requisição do espaço da pilha (*stack*)
- Variáveis dinâmicas: alocadas através de requisição do espaço do *heap*.
  - O heap é a região da memória entre o programa (permanente) e a stack
  - Tamanho do heap é a princípio desconhecido do programa

Memória


Topo da Memória
StackPointer
Início da Pilha
Direção de crescimento
HeapPointer
Início da Área
Alocável
Variáveis estáticas
a
b
Programa executável
Código objeto
10010101...
Constantes
"constante"
Sist.Operacional
Base da Memória

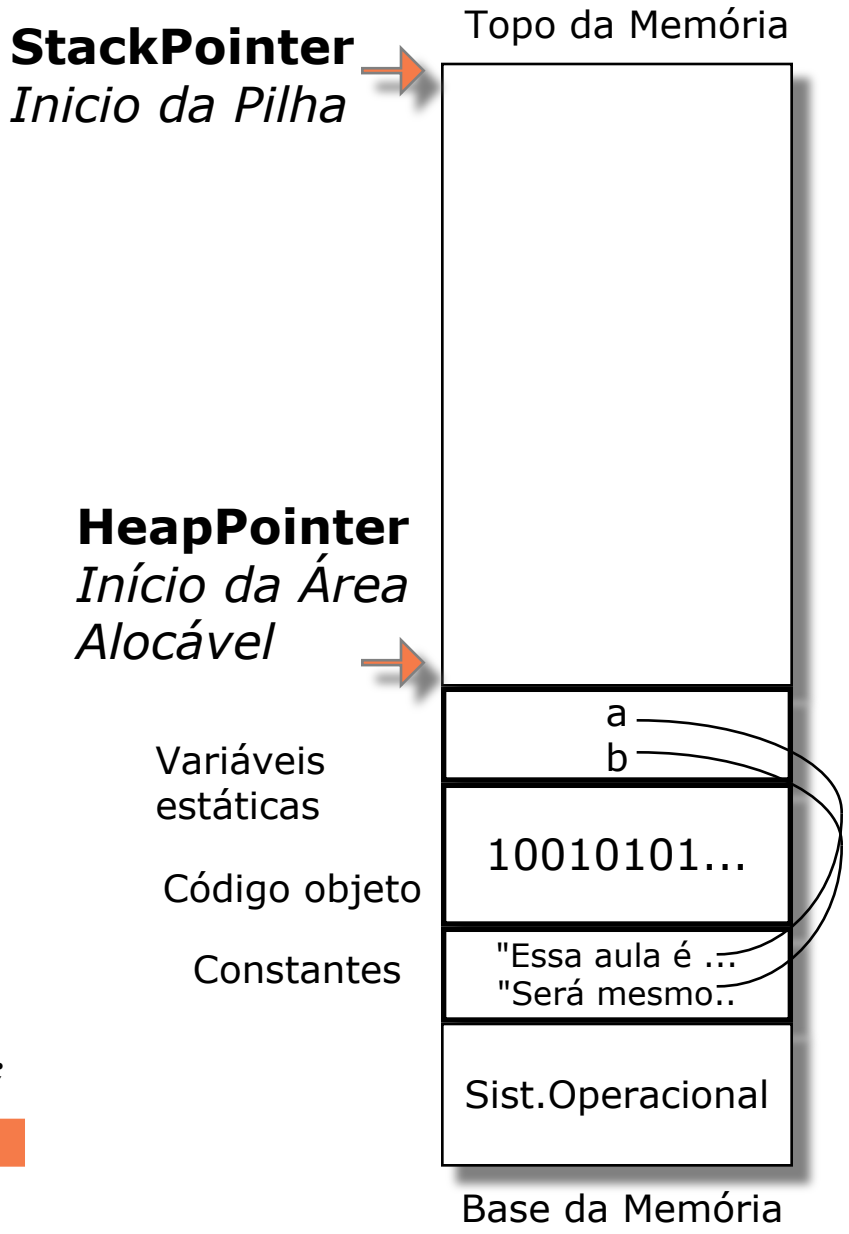
Programa:

```
#include <stdio.h>
char *a, *b, c[4]="sim";

int func_A ()
{
    int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main (...)
{
   a = "Essa aula é legal";
   b = "Será mesmo?"
   func_B();
}
```



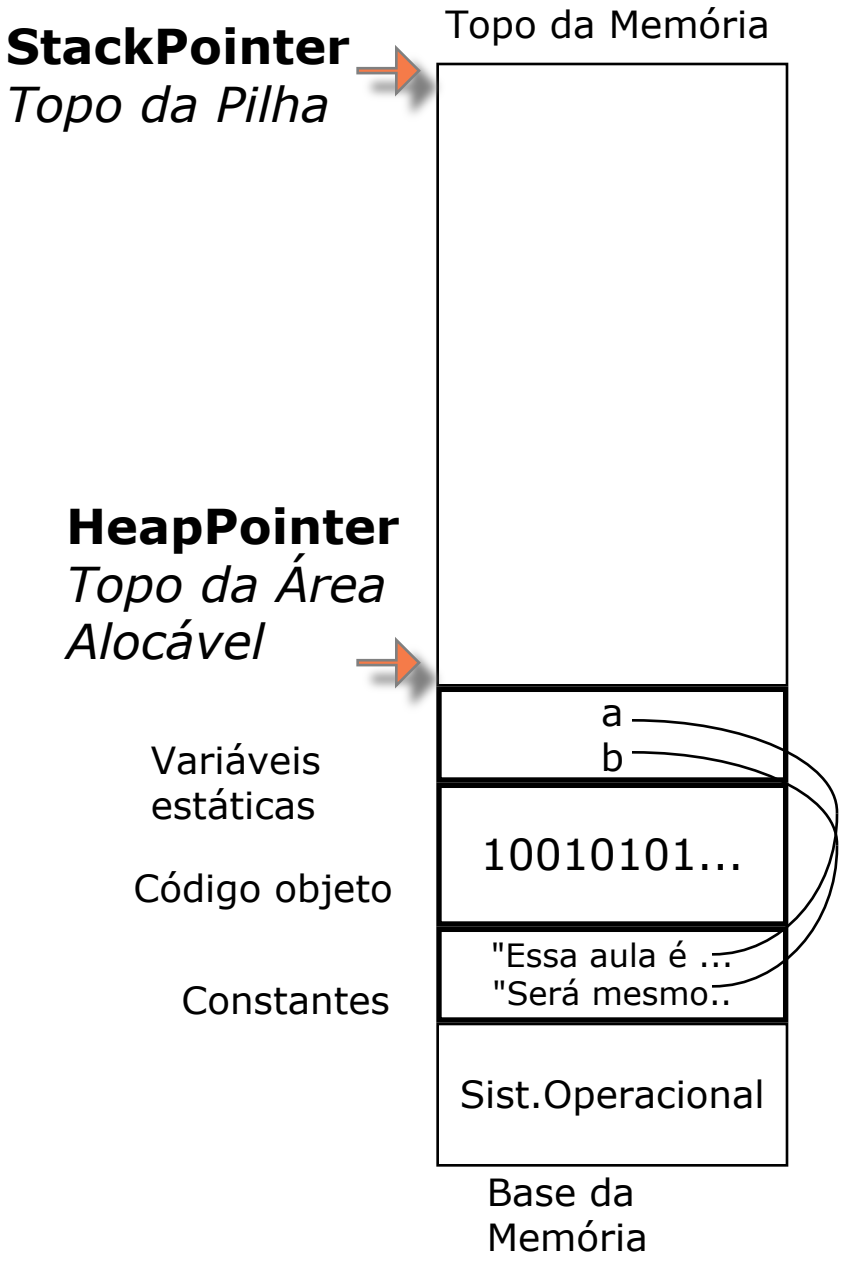
Programa:

```
#include <stdio.h>
char *a, *b;
int func_A ()
{
    int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main (...)
{
   a = "Essa aula é legal";
   b = "Será mesmo?";
   func_B();
}
```



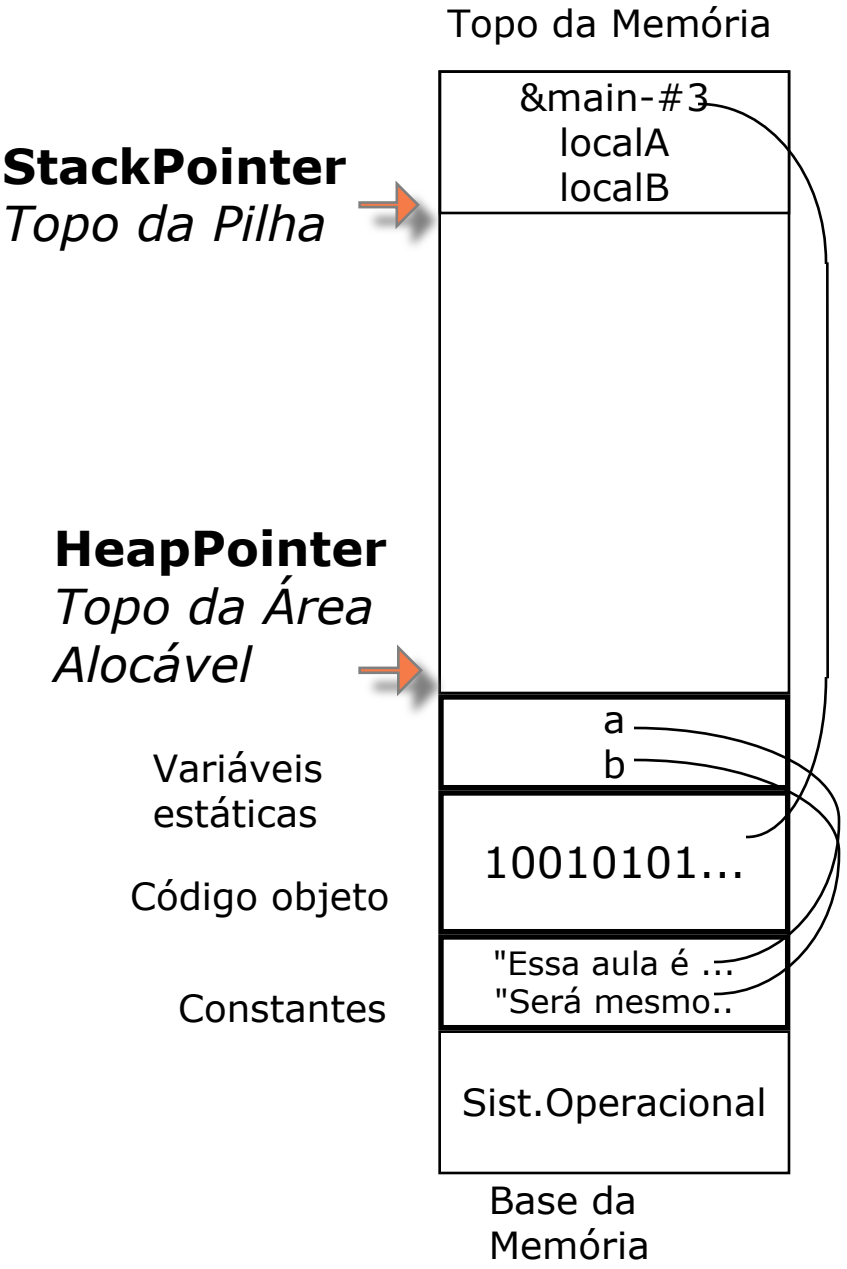
Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
    int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main (...)
{
   a = "Essa aula é legal";
   b = "Será mesmo?"
   func_B();
}
```



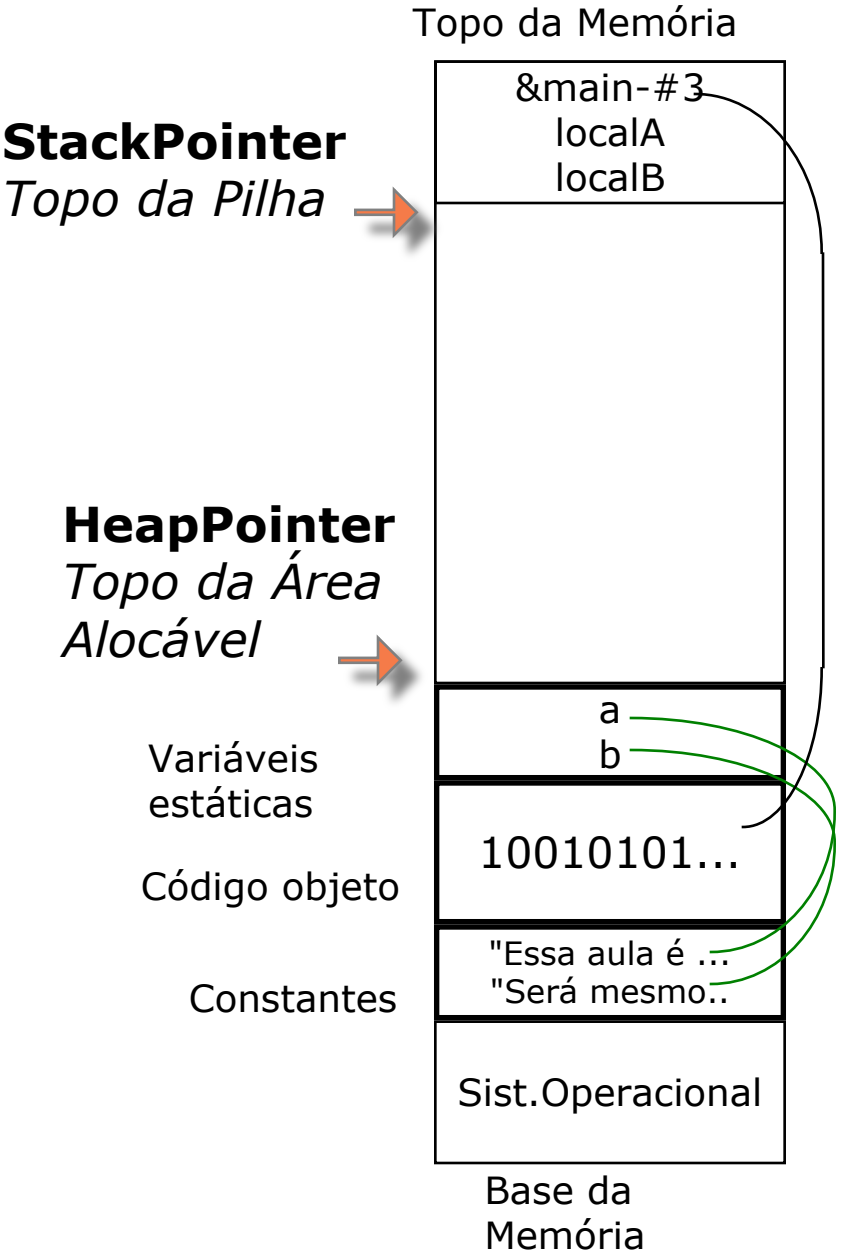
Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
    int local1, local2;
    - - -
}
void func_B ()
{
    int localA, localB;
    localA = func_A();
    localB = func_A();
}
main (...)
{
a = "Essa aula é legal";
b = "Será mesmo?"
func_B();
}
```



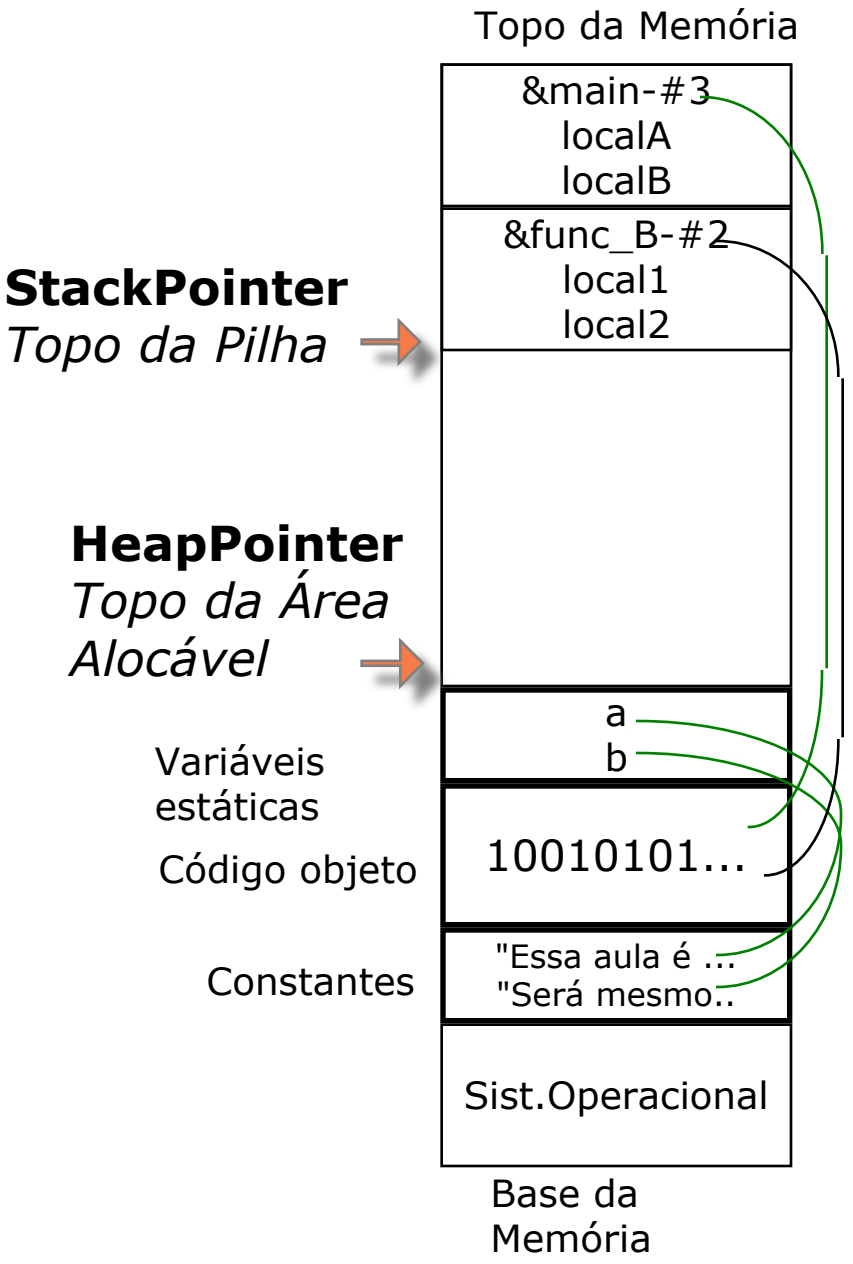
Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
   int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main (...)
{
a = "Essa aula é legal";
b = "Será mesmo?"
func_B();
}
```



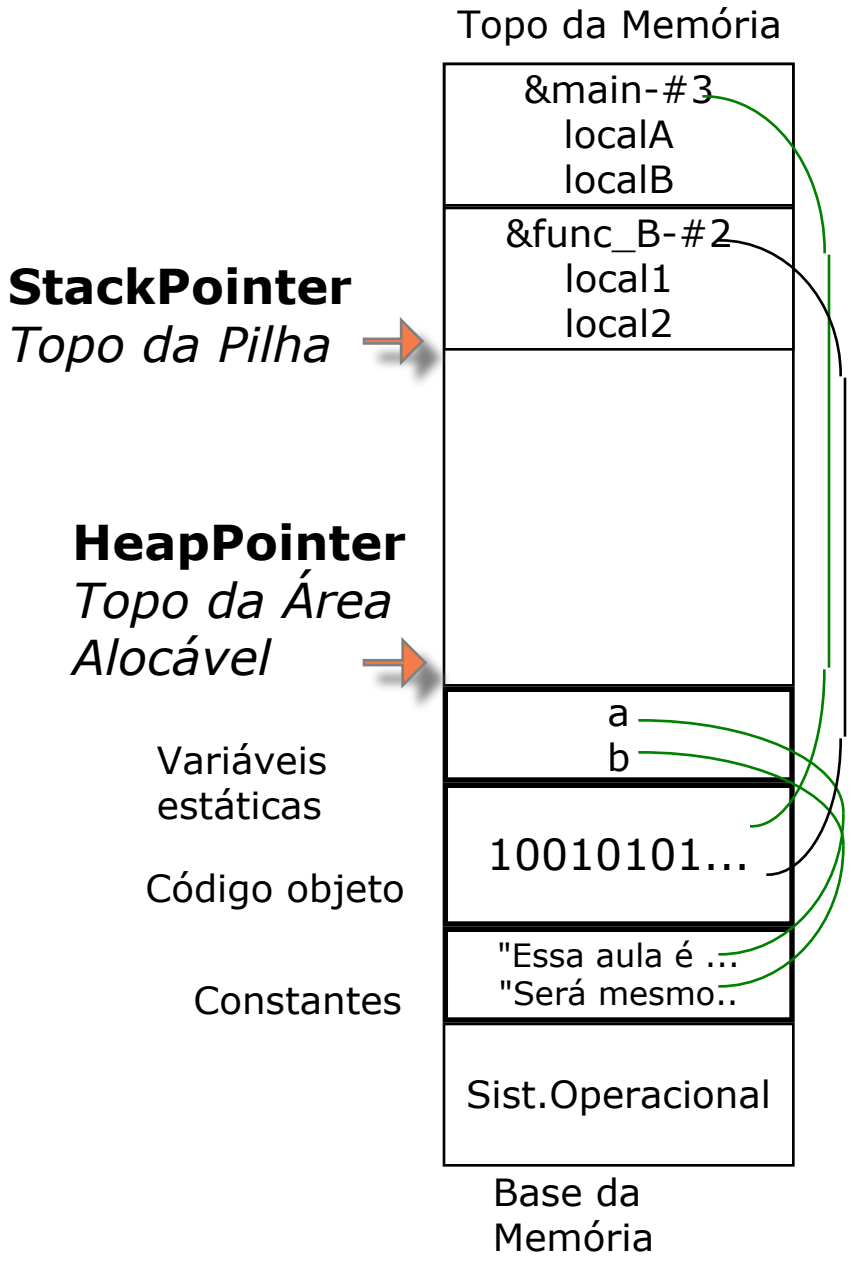
Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
    int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main (...)
{
a = "Essa aula é legal";
b = "Será mesmo?"
func_B();
}
```



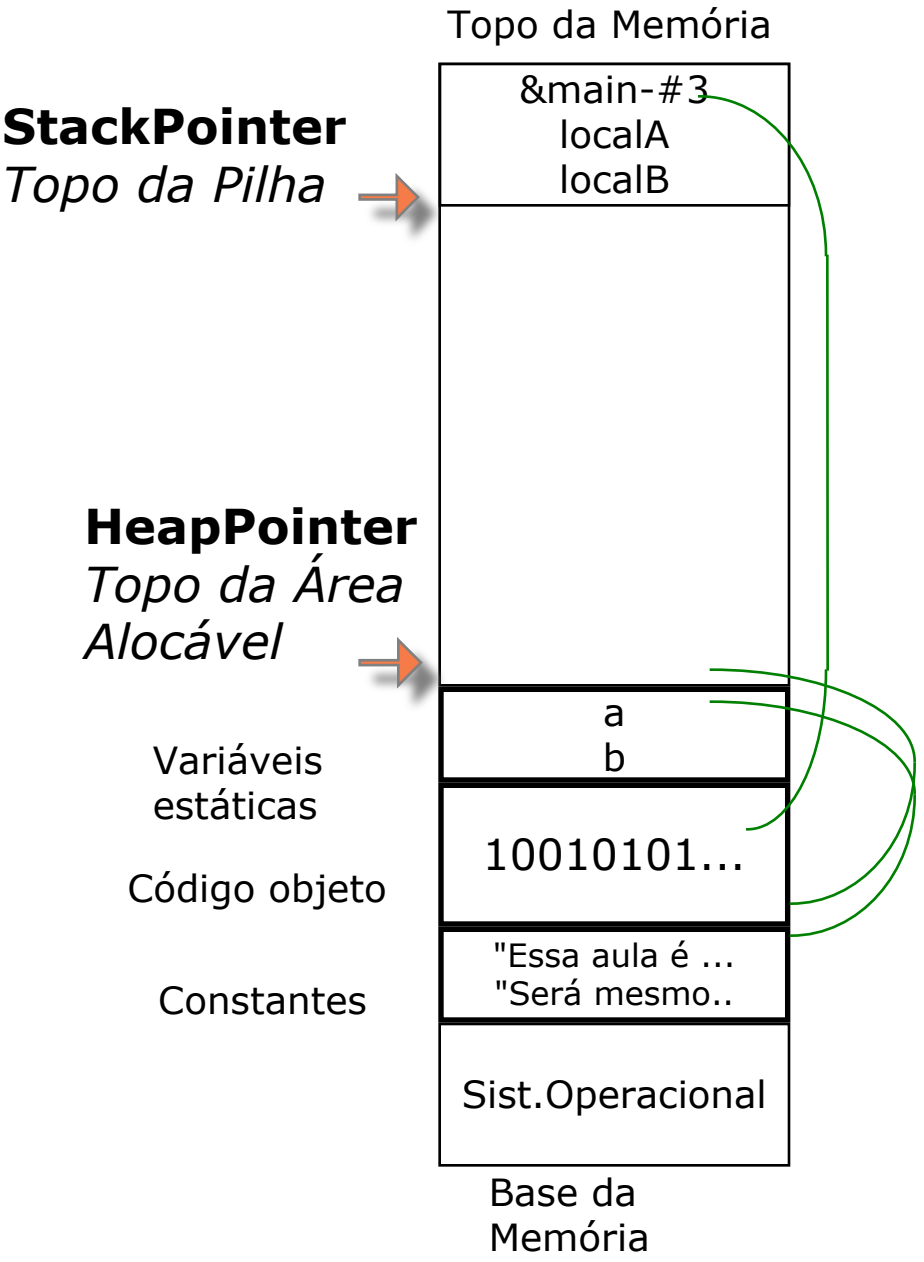
Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
   int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main (...)
{
a = "Essa aula é legal";
b = "Será mesmo?"
func_B();
}
```



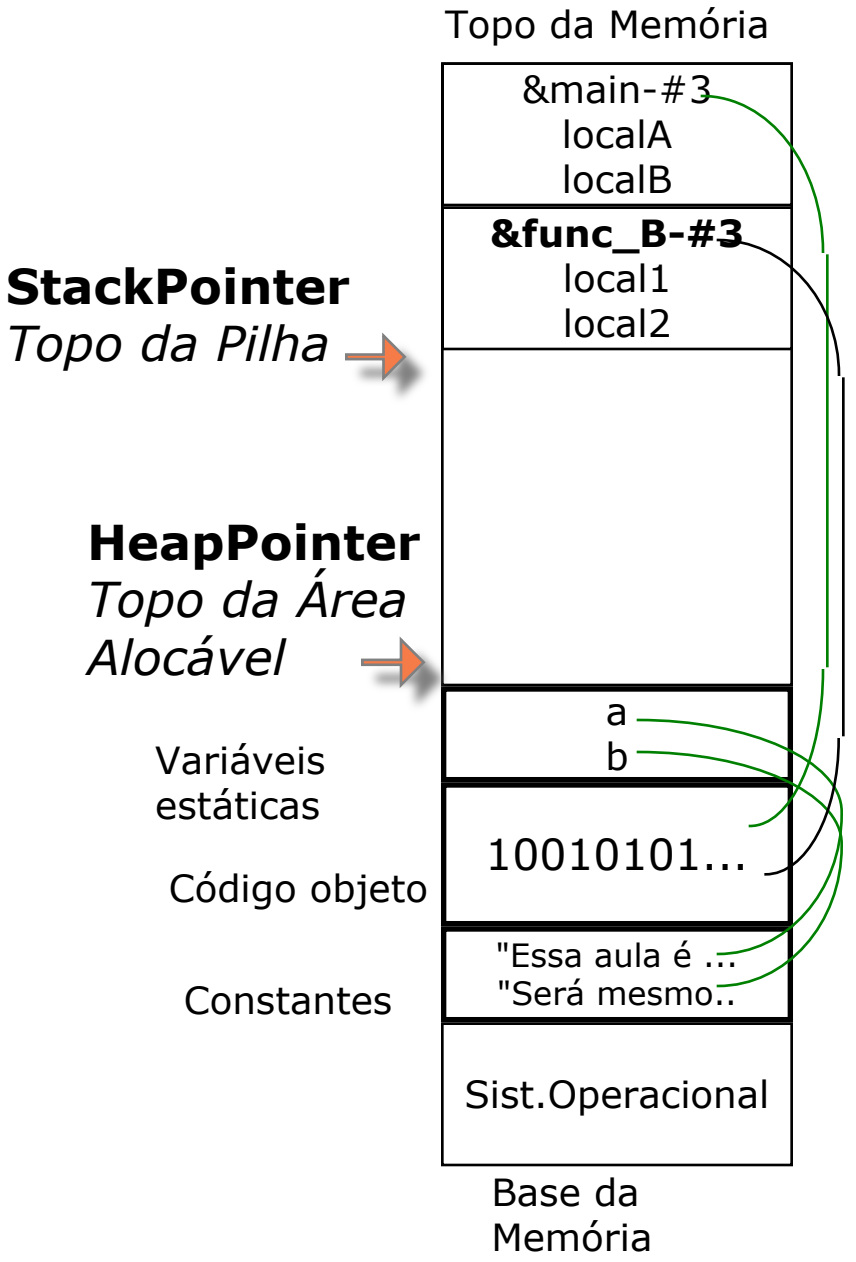
Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
   int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main (...)
{
a = "Essa aula é legal";
b = "Será mesmo?"
func_B();
}
```



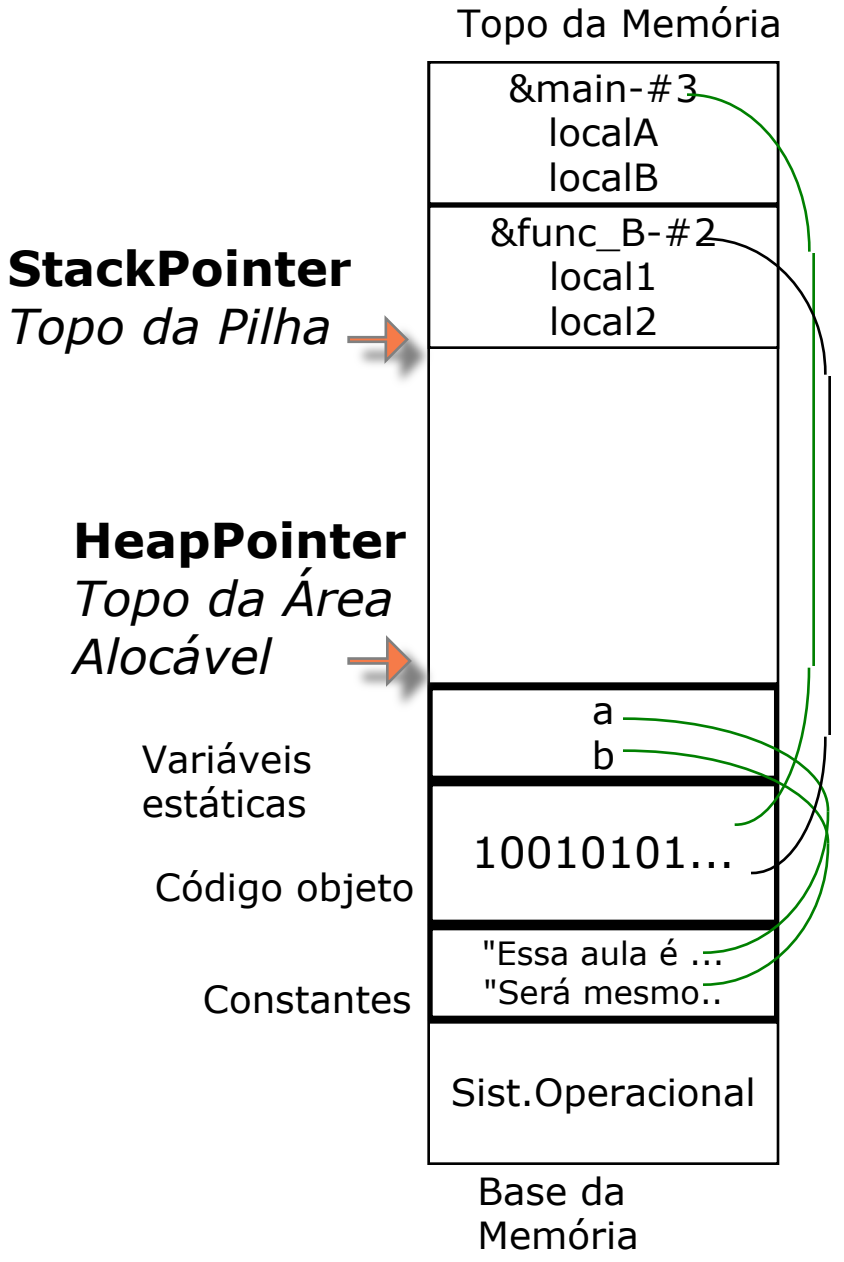
Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
    int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main ()
{
a = "Essa aula é legal";
b = "Será mesmo?"
func_B();
}
```



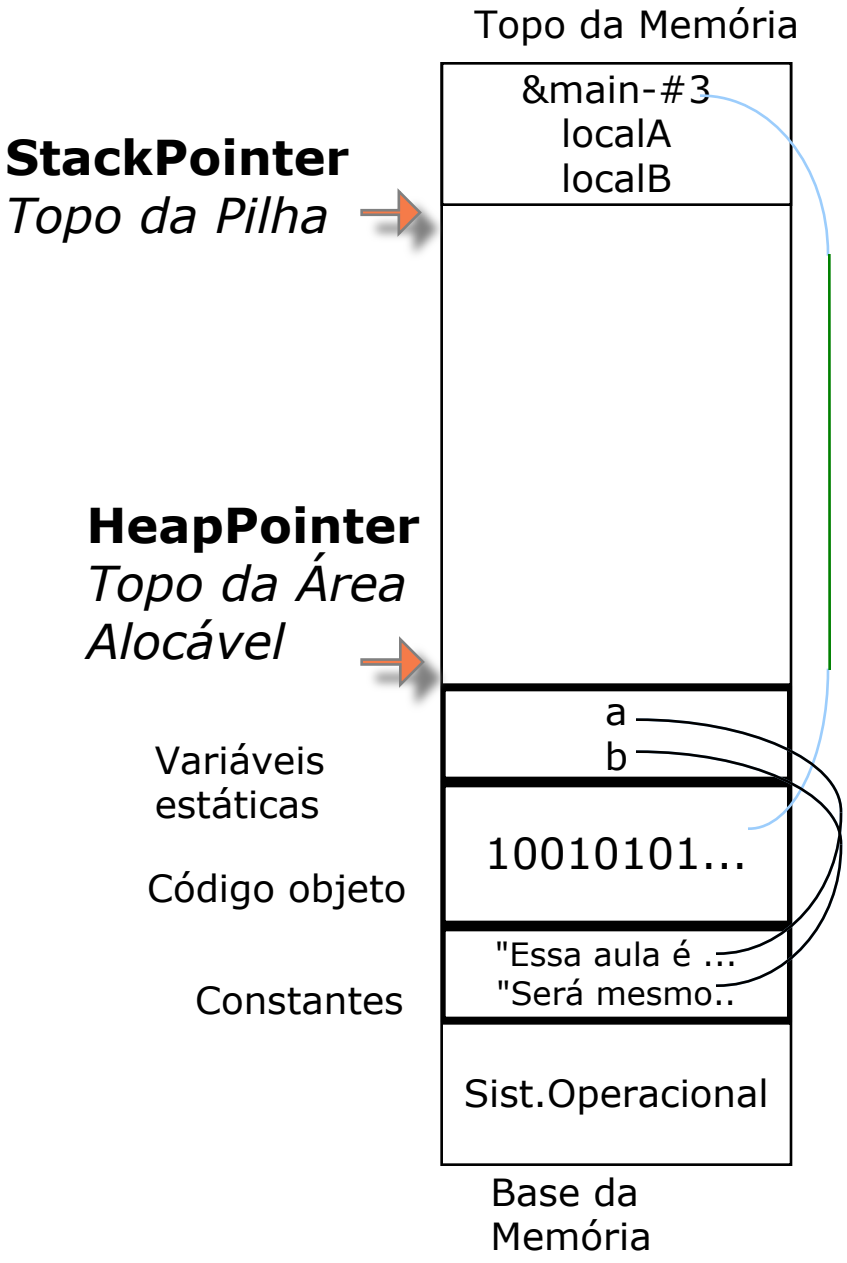
Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
    int local1, local2;
    - - -
}
void func_B ()
{
    int localA, localB;
    localA = func_A();
    localB = func_A();
}
main (...)
{
a = "Essa aula é legal";
b = "Será mesmo?"
func_B();
}
```



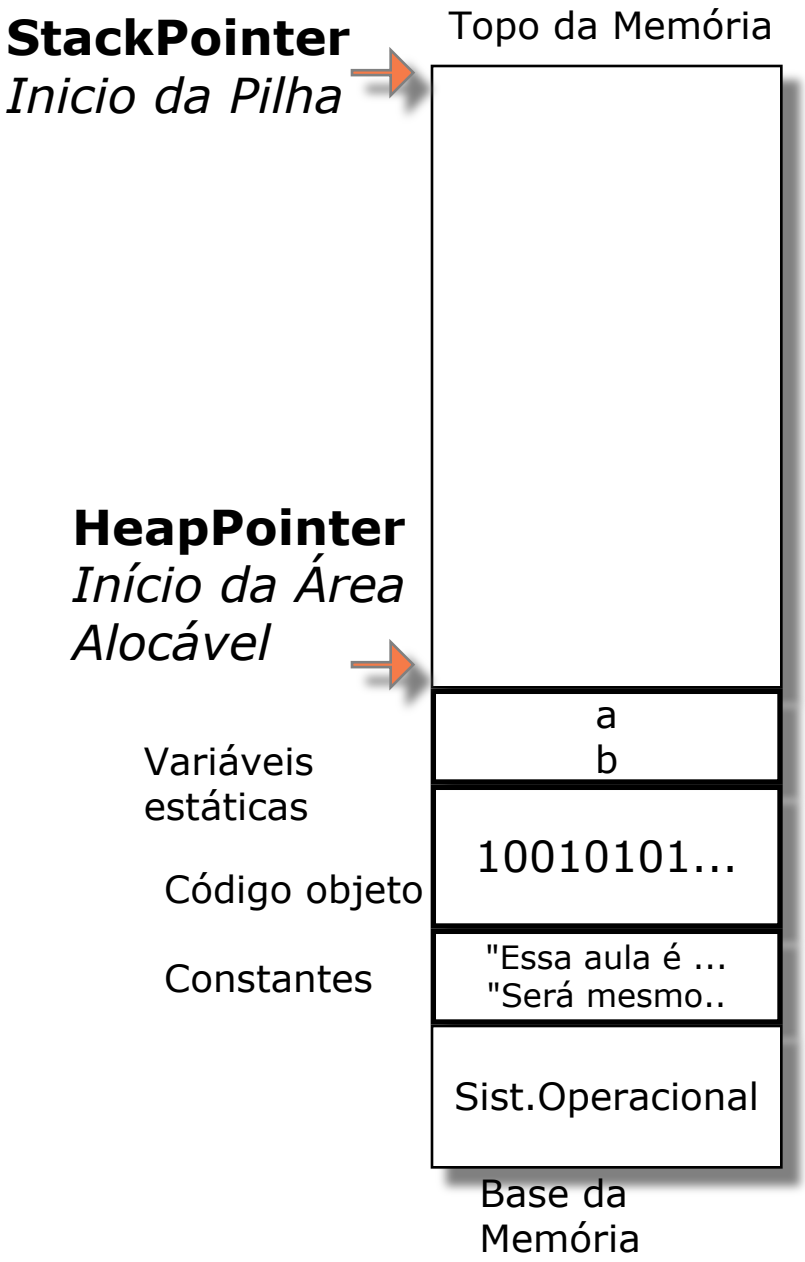
Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
    int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main (...)
{
a = "Essa aula é legal";
b = "Será mesmo?"
func_B();
}
```



Programa:

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
   int local1, local2;
   - - -
}
void func_B ()
{
   int localA, localB;
   localA = func_A();
   localB = func_A();
}
main (...)
{
a = "Essa aula é legal";
b = "Será mesmo?"
func_B();
}
```



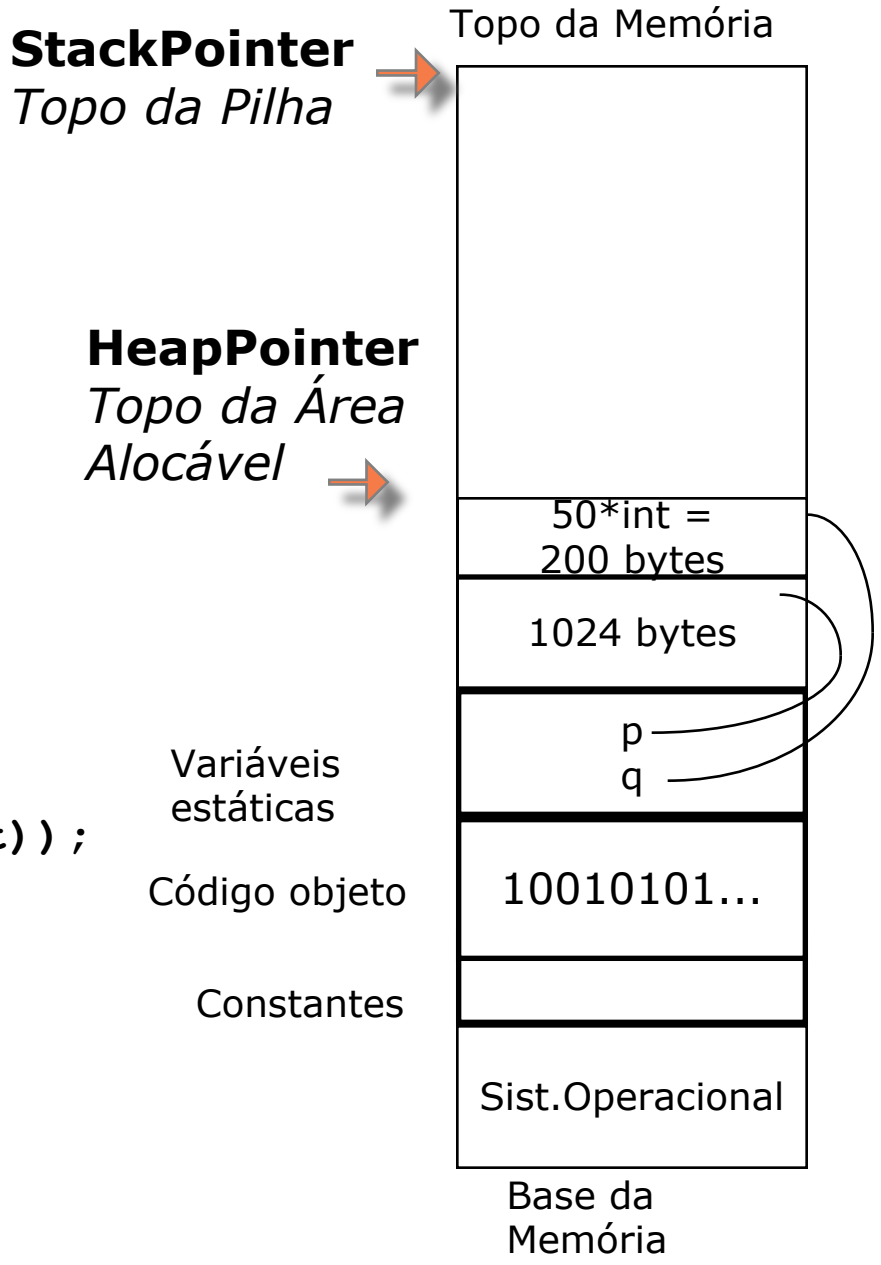
Alocação Dinâmica com malloc

- `void *malloc (tamanho numero_bytes)`
  - Retorna um ponteiro genérico para a área alocada
  - Retorna NULL se não for possível alocar
  - Usar *type casting* para especificar um tipo

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

if (v==NULL) // nao alocou …  if(!v)
    ...
```

Programa:

```
#include <stdlib.h>
#include <stdio.h>

char *p = NULL;
int  *q = NULL;

main (...)
{
  p = (char *) malloc(1024);
       // Aloca 1k
       // bytes de RAM

  q = (int *) malloc(50*sizeof(int));
       // Aloca espaço
       // para 50 inteiros.
}
```



DesAlocando com free

- `void free (void *p)`
  - Devolve a memória previamente alocada e referenciada por p
  - O ponteiro p deve ter sido alocado dinamicamente
  - Se p for NULL ==> sem efeito
  - Definição, mas sem inicialização; PERIGO!

int* p; free(p); ===> ERRO

int* p = NULL; free(p); ===> ok

Desalocando com FREE (sem problemas)

Resolvendo o problema de usar um apontador cuja área de memória já foi desalocada:

```
#define FREE(p) if (p!=NULL) {free(p);p=NULL;}

...
int n, *v = NULL;
   ...
   n = ...
   v = (int*)malloc(n);
   ...
   FREE(v);
   ...
   FREE(v); // redundante, mas sem problema
```

Exemplos com Ponteiros

Ponteiros

- Permitem o armazenamento e manipulação de endereços de memória
- *Forma geral de declaração*
  - ***tipo *nome ou tipo* nome***
  - Símbolo * indica ao compilador que a variável guardará um endereço da memória
  - Neste endereço da memória haverá um valor do tipo especificado por **tipo**
  - **char *p**; (p pode armazenar endereço de memória em que existe um caractere armazenado)
  - **int *v**; (v pode armazenar endereço de memória em que existe um inteiro armazenado)
  - **void *q**; (ponteiro genérico)

Exemplos usando Ponteiros

```
/*variável inteiro*/
int a;

/*variavel ponteiro para inteiro */
int* p;
```

```
/* a recebe o valor 5*/
a = 5;

/* p recebe o endereço de a */
p = &a;
```

```
/*conteúdo de p recebe o valor 6 */
*p = 6;
```

Exemplos usando Ponteiros

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char** argv)
{
  int a;
  int *p = NULL;
  p = &a;
  *p = 2;
  printf("a = %d", a)
  return EXIT_SUCCESS;
}
```

Qual valor é impresso?

Exemplos usando Ponteiros

```
...
int main (...)
{
  int a, b, *p = NULL;
  a = 2;
  *p = 3;
  b = a + (*p);
  printf ("%d", b);
  return 0;
}
```

Há algo errado neste código?

p está apontado para NULL

Exemplos usando Ponteiros

char nome[30];

- nome (sem *) é um ponteiro para caractere que aponta para o primeiro elemento do nome;

```
int v[20], *p;

p = &v[5];
*p = 0;   // equivalente a fazer v[5] = 0
```

```
char nome[30];
char *apontaPraNome;
...
apontaPraNome = nome;  // copia o endereço
```

Aritmética de Ponteiros

Operadores * &

*** indireção**
- **Devolve o valor apontado pelo ponteiro**

& operador de endereço
- **Devolve o endereço na memória de seu operador**

```
int main(...){
    int *aponta;
    int valor1, valor2;
    valor1 = 5;
    aponta = &valor1; // aponta para valor1
    valor2 = *aponta; // equivale a valor2 = valor1
}
```

- **Precedência:** operadores & e * têm precedência maior que outros operadores (com exceção do menos unário)

```
int valor; int *aponta; valor = *aponta++
```

Aritmética de Ponteiros (1)

- Atribuição
  - Atribuição direta entre ponteiros passa o endereço de memória apontado por um para o outro.

```
int *p1, *p2, x;
x = 4;
p1 = &x;
p2 = p1;
```

Aritmética de Ponteiros (2)

- É possível obter o endereço de funções

```
int main (...)
{
  void *pmain = NULL;
  pmain = main;
  printf("pmain=%p\n", pmain );
  ...
```

Aritmética de Ponteiros (3)

- **Adição e subtração**

```
int *p1, *p2, *p3, *p4, x=0;
p1 = &x;
p2 = ++p1;
p3 = p2 + 4;
p4 = p3 - 5;
```

- Neste exemplo, p1, p2 e p3 apontam para endereços de memória que não estão associados com nenhuma variável. Neste caso, expressões do tipo *p1 *p2 e *p3 resultam em **ERRO (possivelmente não perceptível)**.
- O único endereço de memória acessível é o de x.

Aritmética de Ponteiros (4)

- Importante!
  - **As operações de soma e subtração são baseadas no tamanho do tipo base do ponteiro**
  - Ex.: se p1 aponta para 2000, p1 + 2 vai apontar para:
    - 2002, se tipo base do ponteiro é char (1 byte)
    - 2008, se tipo base do ponteiro é int (4 bytes)
  - Ou seja, este exemplo de soma significa que o valor de p1 é adicionado de duas vezes o tamanho do tipo base.

Aritmética de Ponteiros (5)

- No exemplo anterior, se o endereço de x é 1000:
  - p1 recebe o valor 1000 (endereço de memória de x)
  - p2 recebe o valor 1004 e p1 tem seu valor atualizado para 1004.

```
x=0;
p1 = &x;
p2 = ++p1;
p3 = p2 + 4;
p4 = p3 - 5;
```

  - p3 recebe o valor 1004 + 4 * 4 = 1020.
  - p4 recebe o valor 1020 - 5 * 4 = 1000.

- Se o tipo base dos ponteiros acima fosse char* (1 byte), os endereços seriam, respectivamente: 1000, 1001, 1005 e 1000.

Aritmética de Ponteiros (6)

- Explique a diferença entre: (int *p)

  p++; (*p)++; *(p++);

- Comparação entre ponteiros (verifica se um ponteiro aponta para um endereço de memória maior que outro)

```
int *p; *q;
...
if (p < q)
    printf (“p aponta para um endereço menor
                que o de q”);
```

Ponteiros, Vetores e Matrizes

- Ponteiros, vetores e matrizes são muito relacionados em C

- Já vimos que vetores também são ponteiros.
  - **char nome[30]**
  - **nome** sozinho é um ponteiro para caractere, que aponta para a primeira posição do nome

- As seguintes notações são equivalentes:
  - variável[índice]
  - *(variável+índice)
  - variável[0] equivale a *variável

Vetores e Matrizes de Ponteiros

- Ponteiros podem ser declarados como vetores ou matrizes multidimensionais. Exemplo:

```
int *vet[30]; /* Vetor de 30 ponteiros
              /* para números inteiros */
int a=1, b=2, c=3;

vet[0] = &a; /* vet[0] aponta para a*/
vet[1] = &b;
vet[2] = &c;
/* Imprime "a: 1, b: 2"...     */
printf("a: %d, b: %d", *vet[0], *vet[1]);
===> a: 1, b: 2
```

Vetores e Matrizes de Ponteiros (2)

- Importante:
  - Quando alocamos um vetor de ponteiros para inteiros, não necessariamente estamos alocando espaço de memória para armazenar os valores inteiros!

- No exemplo anterior, alocamos espaço de memória para a, b e c (3 primeiras posições do vetor apontam para as posições de memória ocupadas por a, b, e c)

Vetores e matrizes de Ponteiros (3)

- Matrizes de ponteiros são muito utilizadas para manipulação de strings. Por exemplo:

```
char *mensagem[] = {  /* vetor inicializado */
                      "arquivo não encontrado",
                      "erro de leitura",
                      "erro de escrita",
                      "impossível criar arquivo"};
void escreveMensagemDeErro (int num)
{
   printf ("%s\n", mensagem[num]);
}
int main ()
{
   escreveMensagemDeErro( 3 );
}
```

%s imprime a string até encontrar o car. “\0”

Vetores e matrizes de Ponteiros (4)

- Manipular inteiros é um pouco diferente:

```
int *vetor[40];
void imprimeTodos()
{
   for (int i=0; i < 40; i++)
        printf ("%d\n", *vetor[i]);
}
```

- *vetor[i] equivale a **(vetor+i)
- vetor aponta para um ponteiro que aponta para o valor do inteiro
- Indireção Múltipla ou Ponteiros para Ponteiros

Vetores e matrizes de Ponteiros (4)

- Alocando dinamicamente um vetor:

```
int *vetor = (int*)malloc(tam * sizeof(int));
for (int i = 0; i < tam; i++)
    vetor[i] = i;
for (int i = 0; i < tam; i++)
    printf("%d\n", *(vetor+i));
free(vetor);
```

- vetor[i] equivale a *(vetor+i)
- vetor aponta para o endereço do primeiro inteiro contido no vetor

Ponteiros para Ponteiros ou Indireção Múltipla

- Podemos usar ponteiros para ponteiros implicitamente, como no exemplo anterior
- Também podemos usar uma notação mais explícita, da seguinte forma:
  - **tipo **variável;**

- ****variável** é o conteúdo final da variável apontada;

- ***variável** é o conteúdo do ponteiro intermediário.

Ponteiros para Ponteiros

```
#include <stdio.h>
main (...)
{
    int x, *p, **q;
    x = 10;
    p = &x;                    // p aponta para x
    q = &p;                    // q aponta para p
    printf ("%d\n", **q);      // imprime 10...
}
```

Matriz dinâmica usando Ponteiros

- Alocando dinamicamente uma matriz:

```
...
int** mat = (int**)malloc(lins*sizeof(int*));
for (int l = 0; l < lins; l++)
{
   mat[l] = (int*)malloc(cols*sizeof(int));
   for (int c = 0; c < cols; c++)
      scanf("%d", &mat[l][c]);
}
```

Matriz dinâmicas usando Ponteiros (2)

- Desalocando uma matriz alocada dinamicamente:

```
...
int** mat = (int**)malloc(lins*sizeof(int*));
...
for (int l = 0; l < lins; l++)
    free(mat[l]);
free(mat);
```