Estruturas de Dados

ponteiros, passagem de parâmetros,modelos de memória,

Variáveis apontadoras (Ponteiros)

- **Definição**: um ponteiro é uma variável cujo conteúdo é um endereço de memória;
  - este endereço normalmente é a posição de uma outra variável na memória;
  - se uma variável contém o endereço de uma outra, é dito que a primeira variável aponta para a segunda.

Declaração de Ponteiros

A declaração de uma variável do tipo ponteiro (ou apontador) consiste do **tipo base** (aquele para o qual o ponteiro vai apontar), um * e o **nome** da variável.

A forma geral é:

```
tipo *nome;
```

ou

```
tipo* nome;
```

Declaração de Ponteiros

Exemplos:

```
int *contador; //Ponteiro para um inteiro.
char *meuString; //Ponteiro para caracteres.
float *raizQuadrada; //Ponteiro para real.
```

Caso especial:

```
void *simplesPonteiro; //Ponteiro genérico.
```

Declarações que também devolvem ponteiros

```
char nome[30];
```

- **nome** sozinho é também um ponteiro para um array de caracteres, que aponta para o primeiro elemento do array.

Exemplo:

```
main() {
    char nome[30];
    char *apontaPraNome;
    .......
    apontaPraNome = nome; //Só o endereço.
}
```

Operadores de Ponteiros

Existem dois operadores especiais para ponteiros:

* **indireção**

Devolve o valor apontado pelo ponteiro.

& operador de endereço

Devolve o endereço na memória de seu operando.

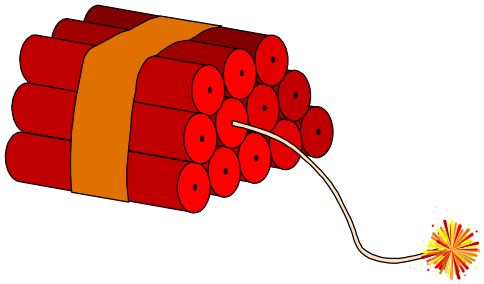
Exemplos

```
main() {
   int *aponta;
   int valor1, valor2;
   valor1 = 5;           // Inicializa valor1 com 5.
   aponta = &valor1;     // aponta recebe o endereço de valor1,
                         // ou seja: passa a apontar para
                         // valor1.
   valor2 = *aponta;     // valor2 recebe o valor apontado por
                         // aponta, nesse caso 5, pois aponta
                         // possui como valor o endereço de
                         // valor1.
}
```

Precedência: tanto o & quanto o * possuem precedência maior do que todos os outros operadores, com exceção dos operadores de incremento / decremento.

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

Aritmética de Ponteiros: expressões envolvendo Ponteiros



- A linguagem "C++" permite que se faça uma série de operações utilizando ponteiros, inclusive várias operações aritméticas - como soma e subtração - além de comparações entre ponteiros;
- isto é muito útil; porém, pode ser também muito perigoso por dar ao programador uma liberdade que em nenhuma outra linguagem de programação (exceto os *assemblers*) é possível.

Atribuição

A atribuição direta entre ponteiros passa o endereço de memória apontado por um para o outro.

```
int *p1, *p2, x;
x = 4;
p1 = &x;              // p1 passa a apontar para x.
p2 = p1;              // p2 recebeu o valor de p1, que é o
                      // endereço de x, ou seja: p2 também
                      // aponta para x.
printf("%p", p2);     // Imprime o endereço de x.
printf("%i", *p2);    // Imprime o valor apontado por p2,
                      // ou seja: o valor de x.
```

O operador de endereço &, quando usado como operador sobre um ponteiro, devolve o endereço ocupado por este ponteiro, não o

endereço apontado por ele!!!

Aritmética de Ponteiros

Duas operações aritméticas são válidas com ponteiros: adição e subtração. Estas são muito úteis com vetores;

a expressão abaixo é válida em "C++":

```
int *p1, *p2, *p3, *p4, x = 0;
 p1 = &x;
 p2 = p1++;
 p3 = p2 + 4;
 p4 = p3 - 5;  // p4 acaba tendo o mesmo valor que p1
               // no começo. Note que p1 foi
               // incrementado e agora tem o
               // valor (&x + 1).
```

Observe que aqui as expressões *p2 e *p3 vão resultar em um erro, já que estes ponteiros estarão apontando para áreas de memória que não estão associadas com nenhuma variável. O único endereço de memória acessível é o de x.

Aritmética de Ponteiros

Para o cálculo do incremento ou decremento é usado sempre o **TAMANHO DO TIPO BASE DO PONTEIRO**.

Isto significa que se p1 aponta para o endereço 2000, p1 + 2 não necessariamente vai ser igual a 2002. Se o tipo base é um inteiro (int *p1), que em Unix sempre possui 4 bytes de tamanho, então p1 + 2 é igual a 2008; ou seja: o valor de p1 adicionado de duas vezes o tamanho do tipo base.

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

Se as variáveis acima fossem do tipo char e char* (1 byte de tipo base), os endereços seriam, respectivamente: 1000, 1001, 1001, 1005 e 1000.

Comparações entre Ponteiros

Você pode comparar ponteiros para saber se um ponteiro aponta para um endereço de memória mais alto do que outro. Exemplo:

```
int *p, *q;
....
if (p < q) {
     printf("p aponta para um endereço menor que o de q");
}
```

Testes como este podem ser úteis em programas que utilizam vetores e matrizes.

Exercício: para fazer em casa

Reimplemente o seu programa de pilha com vetor usando como TOPO um ponteiro para o ultimo endereço, que você incrementa, decrementa e testa para saber se a pilha está cheia ou vazia;

para resolver:

modifique a estrutura Pilha da seguinte forma:

```
constantes MAXPILHA = 100;
classe Pilha {
        T dados[MAXPILHA];
        T *topo;
};
```

Exercício 2: para fazer em casa

Modifique os algoritmos de manipulação da pilha de forma que se utilize ponteiros para referenciar os elementos da pilha.

Exemplo:

```
T MÉTODO empilha(T dado)
início
        SE (pilhaCheia) ENTÃO
             THROW(ErroPilhaCheia)
        SENÃO
                // Se houver espaço, incremento o
                // ponteiro topo e faço o valor
                // apontado por topo receber o novo
                // dado.
                topo <- topo + 1;
                *(topo) <- dado;
                RETORNE(topo);
        FIM SE
fim;
```

Exercício 3: para fazer em casa

Lembre-se de adaptar a inicialização da pilha e também os testes de pilha cheia e vazia. Exemplos:

```
MÉTODO inicializaPilha()
início
        // Fazemos o topo apontar para um endereço de memória
        // anterior ao início do vetor dados para simbolizar
        // que a pilha está vazia.
        topo <- dados - 1;
fim;
Booleano MÉTODO pilhaVazia()
início
        SE (topo < dados) ENTÃO
                // O topo está apontando para um endereço de
                // memória anterior ao próprio início da
                // pilha. Segundo a nossa definição, isto
                // significa que a pilha está vazia.
                RETORNE(Verdadeiro)
        SENÃO
                RETORNE(Falso);
fim;
```

Ponteiros e Matrizes

Ponteiros, Vetores e Matrizes possuem uma relação muito estreita em "C++"

a qual podemos aproveitar de muitas formas para escrever programas que ninguém entende...

A seguir veremos um exemplo.

```
char nome[30] = "José da Silva";
char *p1, *p2;
char car;
int i;

p1 = nome;                          // nome sozinho é um ponteiro
                                    // para o 1° elemento de nome[].
car = nome[3];                      // Atribui 'é' a car.
car = p1[0];                        // Atribui 'J' a car. Válido.
p2 = &nome[5];                      // Atribui a p2 o endereço da 6ª
                                    // posição de nome, no caso 'd'.
printf("%s", p2);                   // Imprime "da Silva"...

p2 = p1;                            // Evidentemente válido.
p2 = p1 + 5;                        // Equivalente a p2 = &nome[5]
printf("%s", (p1 + 5));             // Imprime "da Silva"...
printf("%s", (p1 + 20));            // Cuidado: imprime lixo!!!

for(i = 0;i <= strlen(nome)-1;i++) {
   printf("%c", nome[i]);           // Imprime 'J','o','s'...
   p2 = p1 + i;
   printf("%c", *p2);               // Imprime 'J','o','s'...
}
```

```
char nome[30] = "José da Silva";
char *p1, *p2;
char car;
int i;

p1 = nome;                          // nome sozinho é um ponteiro
                                    // para o 1° elemento de nome[].
car = nome[3];                      // Atribui 'é' a car.
car = p1[0];                        // Atribui 'J' a car. Válido.
p2 = &nome[5];                      // Atribui a p2 o endereço da 6ª
                                    // posição de nome, no caso 'd'.
printf("%s", p2);                   // Imprime "da Silva"...

p2 = p1;                            // Evidentemente válido.
p2 = p1 + 5;                        // Equivalente a p2 = &nome[5]
printf("%s", (p1 + 5));             // Imprime "da Silva"...
printf("%s", (p1 + 20));            // Cuidado: imprime lixo!!!

for(i = 0;i <= strlen(nome)-1;i++) {
   printf("%c", nome[i]);           // Imprime 'J','o','s'...
   p2 = p1 + i;
   printf("%c", *p2);               // Imprime 'J','o','s'...
}
```

```
char nome[30] = "José da Silva";
char *p1, *p2;
char car;
int i;

p1 = nome;                          // nome sozinho é um ponteiro
                                    // para o 1° elemento de nome[].
car = nome[3];                      // Atribui 'é' a car.
car = p1[0];                        // Atribui 'J' a car. Válido.
p2 = &nome[5];                      // Atribui a p2 o endereço da 6ª
                                    // posição de nome, no caso 'd'.
printf("%s", p2);                   // Imprime "da Silva"...

p2 = p1;                            // Evidentemente válido.
p2 = p1 + 5;                        // Equivalente a p2 = &nome[5]
printf("%s", (p1 + 5));             // Imprime "da Silva"...
printf("%s", (p1 + 20));            // Cuidado: imprime lixo!!!

for(i = 0;i <= strlen(nome)-1;i++) {
   printf("%c", nome[i]);           // Imprime 'J','o','s'...
   p2 = p1 + i;
   printf("%c", *p2);               // Imprime 'J','o','s'...
}
```

```
char nome[30] = "José da Silva";
char *p1, *p2;
char car;
int i;

p1 = nome;                          // nome sozinho é um ponteiro
                                    // para o 1° elemento de nome[].
car = nome[3];                      // Atribui 'é' a car.
car = p1[0];                        // Atribui 'J' a car. Válido.
p2 = &nome[5];                      // Atribui a p2 o endereço da 6ª
                                    // posição de nome, no caso 'd'.
printf("%s", p2);                   // Imprime "da Silva"...

p2 = p1;                            // Evidentemente válido.
p2 = p1 + 5;                        // Equivalente a p2 = &nome[5]
printf("%s", (p1 + 5));             // Imprime "da Silva"...
printf("%s", (p1 + 20));            // Cuidado: imprime lixo!!!

for(i = 0;i <= strlen(nome)-1;i++) {
   printf("%c", nome[i]);           // Imprime 'J','o','s'...
   p2 = p1 + i;
   printf("%c", *p2);               // Imprime 'J','o','s'...
}
```

```
char nome[30] = "José da Silva";
char *p1, *p2;
char car;
int i;

p1 = nome;                          // nome sozinho é um ponteiro
                                    // para o 1° elemento de nome[].
car = nome[3];                      // Atribui 'é' a car.
car = p1[0];                        // Atribui 'J' a car. Válido.
p2 = &nome[5];                      // Atribui a p2 o endereço da 6ª
                                    // posição de nome, no caso 'd'.
printf("%s", p2);                   // Imprime "da Silva"...

p2 = p1;                            // Evidentemente válido.
p2 = p1 + 5;                        // Equivalente a p2 = &nome[5]
printf("%s", (p1 + 5));             // Imprime "da Silva"...
printf("%s", (p1 + 20));            // Cuidado: imprime lixo!!!

for(i = 0;i <= strlen(nome)-1;i++) {
   printf("%c", nome[i]);           // Imprime 'J','o','s'...
   p2 = p1 + i;
   printf("%c", *p2);               // Imprime 'J','o','s'...
}
```

```
char nome[30] = "José da Silva";
char *p1, *p2;
char car;
int i;

p1 = nome;                          // nome sozinho é um ponteiro
                                    // para o 1° elemento de nome[].
car = nome[3];                      // Atribui 'é' a car.
car = p1[0];                        // Atribui 'J' a car. Válido.
p2 = &nome[5];                      // Atribui a p2 o endereço da 6ª
                                    // posição de nome, no caso 'd'.
printf("%s", p2);                   // Imprime "da Silva"...

p2 = p1;                            // Evidentemente válido.
p2 = p1 + 5;                        // Equivalente a p2 = &nome[5]
printf("%s", (p1 + 5));             // Imprime "da Silva"...
printf("%s", (p1 + 20));            // Cuidado: imprime lixo!!!

for(i = 0;i <= strlen(nome)-1;i++) {
   printf("%c", nome[i]);           // Imprime 'J','o','s'...
   p2 = p1 + i;
   printf("%c", *p2);               // Imprime 'J','o','s'...
}
```

```
char nome[30] = "José da Silva";
char *p1, *p2;
char car;
int i;

p1 = nome;                          // nome sozinho é um ponteiro
                                    // para o 1° elemento de nome[].
car = nome[3];                      // Atribui 'é' a car.
car = p1[0];                        // Atribui 'J' a car. Válido.
p2 = &nome[5];                      // Atribui a p2 o endereço da 6ª
                                    // posição de nome, no caso 'd'.
printf("%s", p2);                   // Imprime "da Silva"...

p2 = p1;                            // Evidentemente válido.
p2 = p1 + 5;                        // Equivalente a p2 = &nome[5]
printf("%s", (p1 + 5));             // Imprime "da Silva"...
printf("%s", (p1 + 20));            // Cuidado: imprime lixo!!!
```

```
for(i = 0;i <= strlen(nome)-1;i++) {
   printf("%c", nome[i]);           // Imprime 'J','o','s'...
   p2 = p1 + i;
   printf("%c", *p2);               // Imprime 'J','o','s'...
}
```

Matrizes de Ponteiros

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

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

vetor[0] = &a;   // vetor[0] passa a apontar para a.
vetor[1] = &b;
vetor[2] = &c;
printf("a: %i, b: %i", *vetor[0], *vetor[1]);
```

Importantíssimo: note que o fato de você alocar um vetor de ponteiros para inteiros não implica que você alocou espaço de memória para armazenar os valores desses inteiros:

- a operação acima foi possível porque com a declaração de a, b e c este espaço foi alocado;
- as posições 0, 1 e 2 do vetor só apontam para as posições de memória ocupadas por a, b e c.

Ponteiros para Ponteiros e Indireção Múltipla

Matrizes de ponteiros são normalmente utilizadas para a manipulação de coleções de strings.

Suponhamos a seguinte função que exibe uma mensagem de erro com base em um código de erro:

```
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]);
}
main () {
        escreveMensagemDeErro(3);
}
```

Ponteiros para Ponteiros e Indireção Múltipla

- Se quiséssemos fazer o mesmo com inteiros, por exemplo, em uma rotina que imprime todos os valores apontados por um vetor de inteiros, já seria diferente:

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

- Você pode ter um ponteiro apontando para outro ponteiro que por sua vez aponta para um valor;
- esta situação é chamada de **Indireção Múltipla** ou de Ponteiros para Ponteiros.

Indireção Múltipla

- Uma forma de declarar ponteiros para ponteiros é a forma implícita já vista antes;
- outra forma que podemos utilizar, quando não sabemos de antemão o espaço em memória a ser utilizado, é de declarar um ponteiro explicitamente como sendo de indireção:

```
main() {
int x, *p, **q;  // q é um ponteiro para
                                  // um ponteiro a inteiro.
x = 10;
p = &x;          // p aponta para x.
q = &p;          // q aponta para p.
printf("%i\n", **q);  // Imprime 10...
}
```

Passagem de Parâmetros usando Ponteiros

```
char *a = "Bananarama";
char b[80] = "uma coisa qualquer";
char *c[5];

void teste1(char *d[]) {  // Recebe vetor de ponteiros para caracter de tamanho
                          // indefinido.
   printf("Teste1: d[0]:%s e d[1]:%s\n\n", d[0], d[1]);
}

void teste2(char **d) {  // Recebe ponteiro para ponteiro para caracter.
   printf("Teste2: d[0]:%s e d[1]:%s\n", d[0], d[1]);
   printf("Teste3: d[0]:%s e d[1]:%s\n", *d, *(d + 1));
}

main() {
   c[0] = a;
   c[1] = b;
   printf("a: %s e b: %s\n\n", a, b);
   printf("c[0]: %s e c[1]: %s\n\n", c[0], c[1]);
   teste1(c);
   teste2(c);
}
```

Passagem de Parâmetros

Existem basicamente três tipos de formas de passagem de parâmetros para um função:

por **valor:**

- quando *copiamos* o valor de uma variável para dentro do parâmetro de uma função;

por **referência:**

- quando passamos para uma função uma *referência* a uma região de memória onde está o valor desta variável;

por **nome:**

- quando passamos para uma função o nome de uma variável, que está em algum lugar e contém o valor.
  - Usada somente em LISP e algumas antigas implementações de ALGOL. Sem interesse para nós.

Passagem de Parâmetros: Modelo de Memória

- Para entendermos as nuances da passagem de parâmetros de forma fundamentada, temos primeiro que entender o **Modelo de Memória** de um computador;
- com isto poderemos entender qual a diferença entre uma variável **local**, uma variável **global** e memória alocada **dinamicamente**.

Passagem de Parâmetros: Modelo de Memória

Para entendermos o modelo de memória, vamos nos basear no modelo mais simples:



As Funções de Alocação Dinâmica de Memória em "C++"

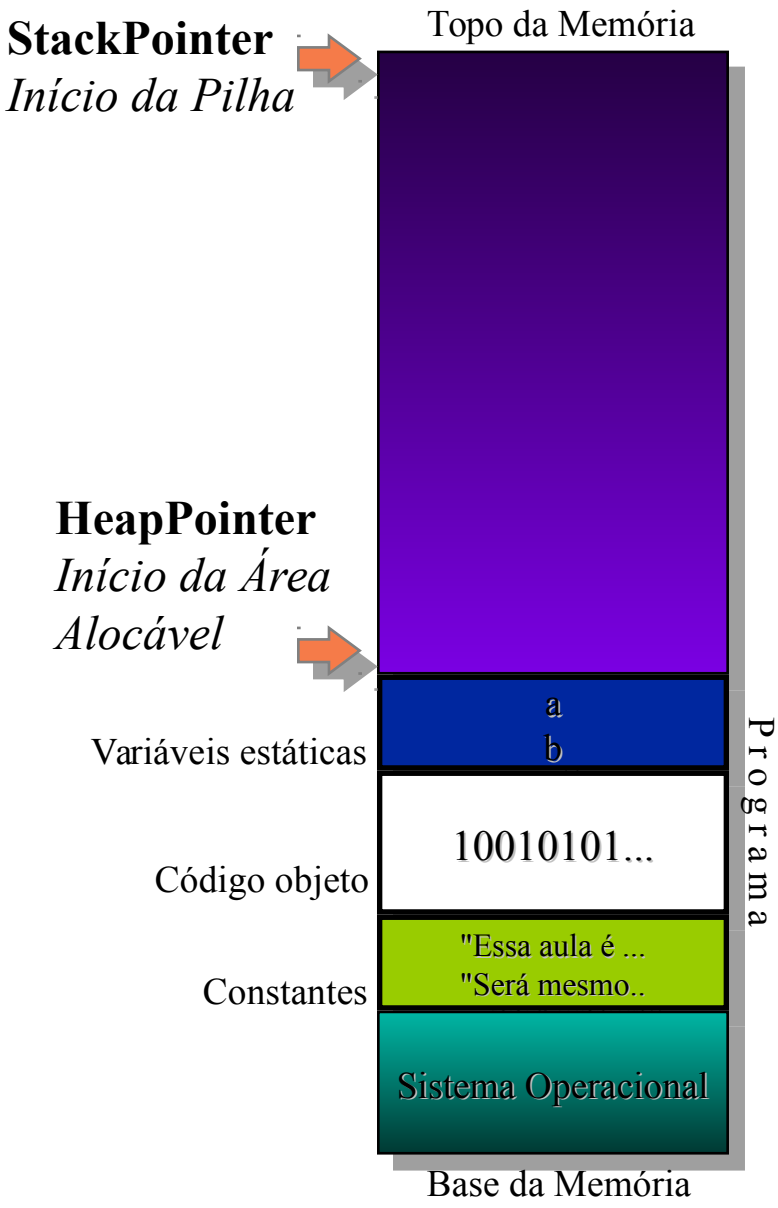
- ***Alocação Dinâmica*** é um meio pelo qual o programa pode obter memória enquanto está em execução.
- Já visto até agora:
  - **Constantes** são "*codificadas*" dentro do código objeto de um programa em tempo de compilação.
  - **Variáveis globais** (*estáticas*) têm a sua alocação codificada em tempo de compilação e são alocadas logo que um programa inicia a execução.
  - **Variáveis locais** em funções (ou *métodos*) são alocadas através da requisição de espaço na pilha (*stack*).

```
#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();
}
```

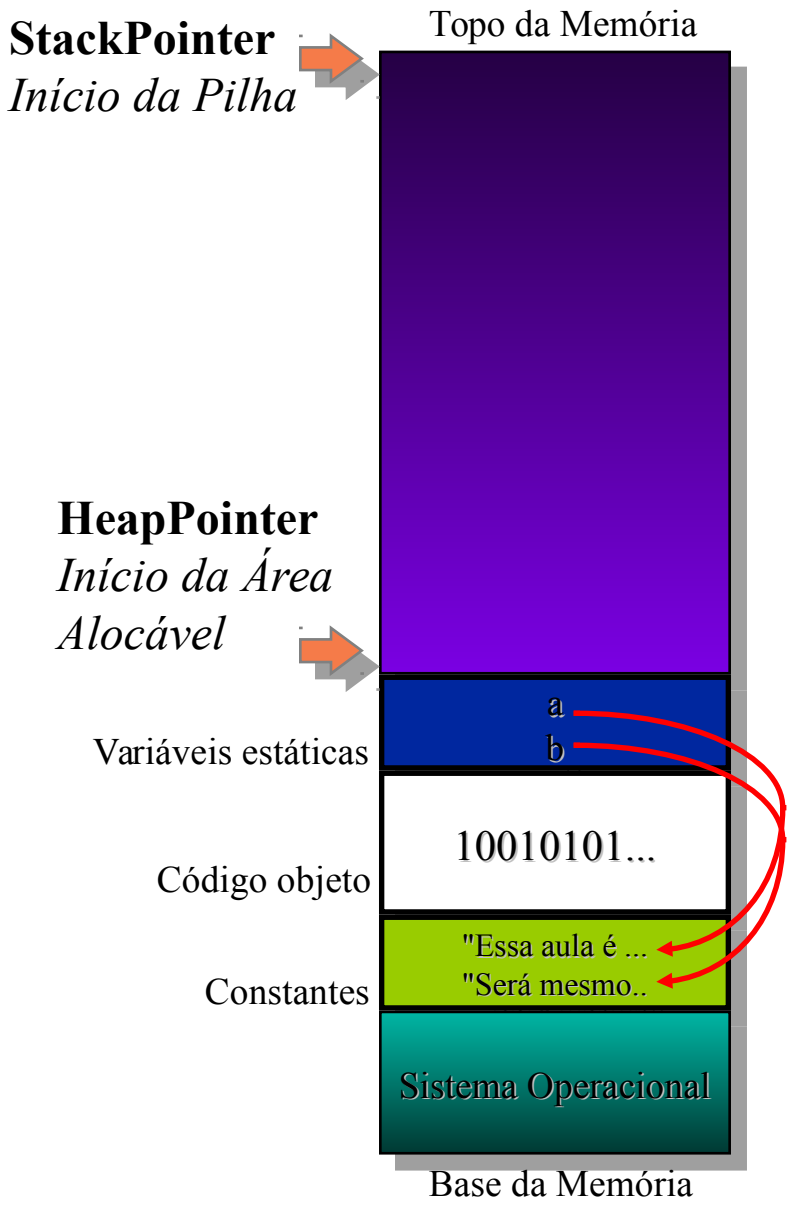


```
#include <stdio.h>
char *a, *b;

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();
}
```



```
#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();
}
```

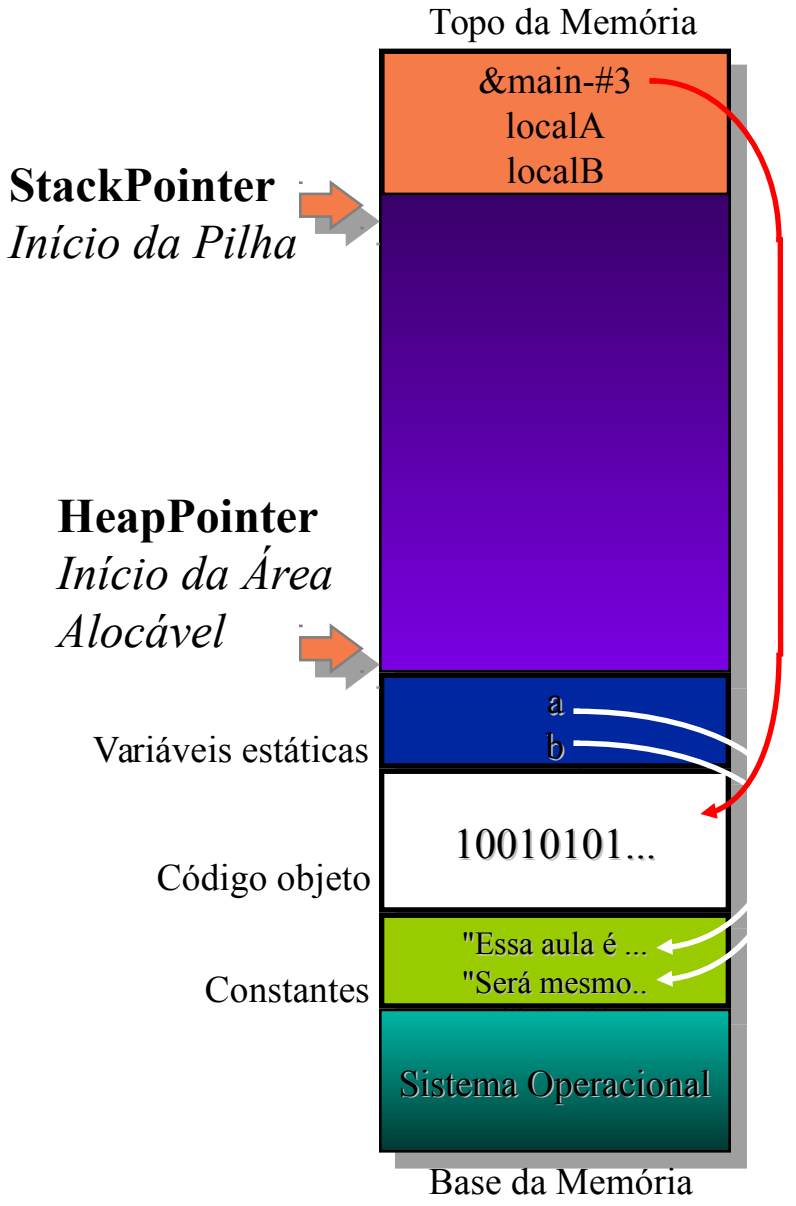


```
#include <stdio.h>
char *a, *b;

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();
}
```

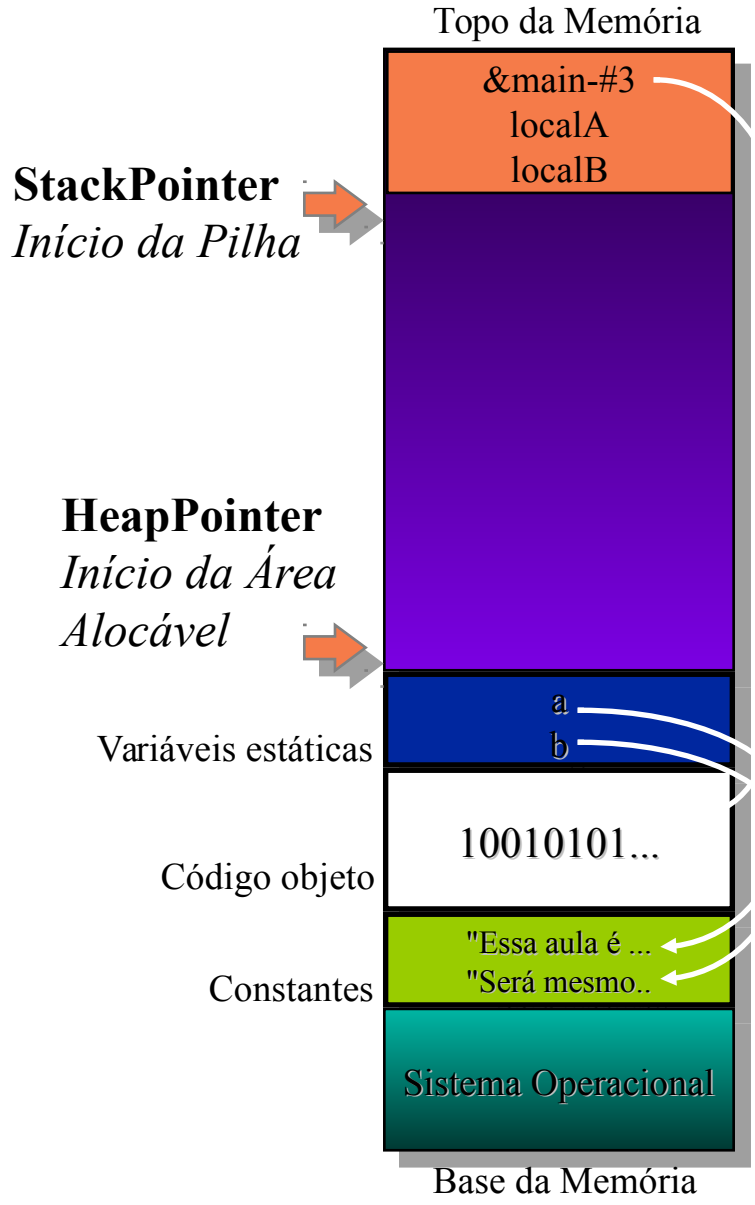


```
#include <stdio.h>
char *a, *b;

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();
}
```

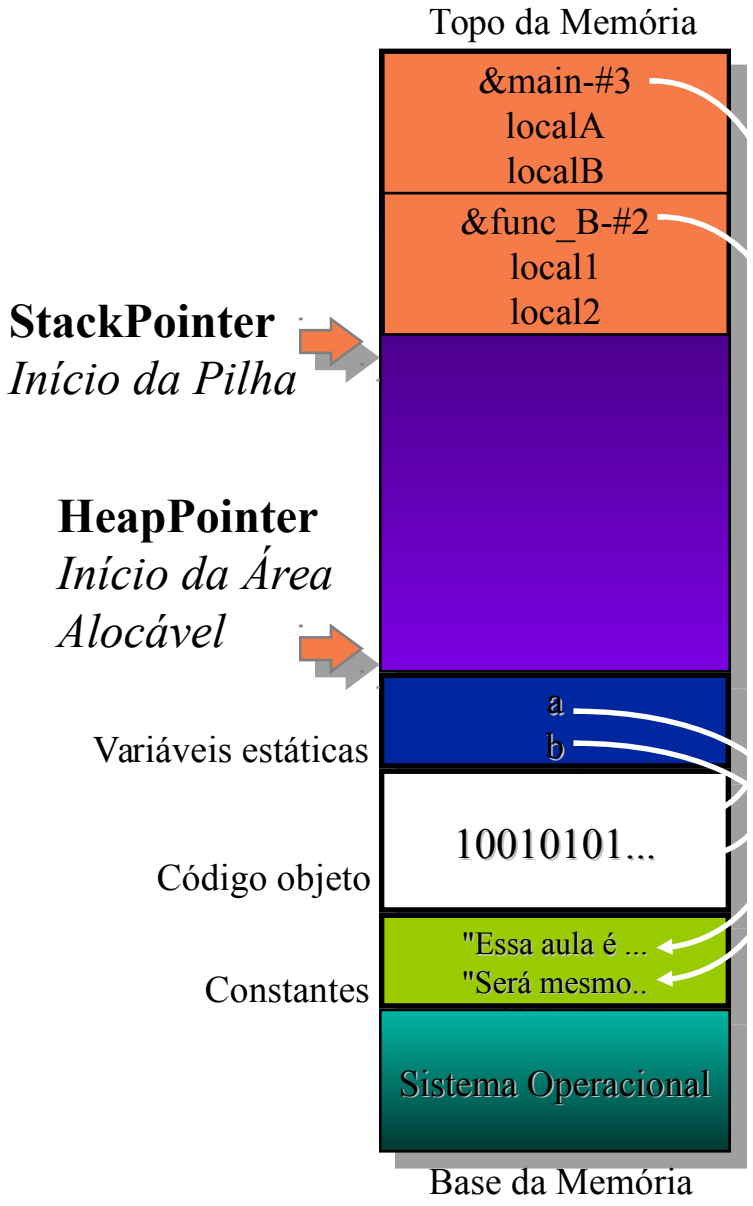


```
#include <stdio.h>
char *a, *b;

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();
}
```

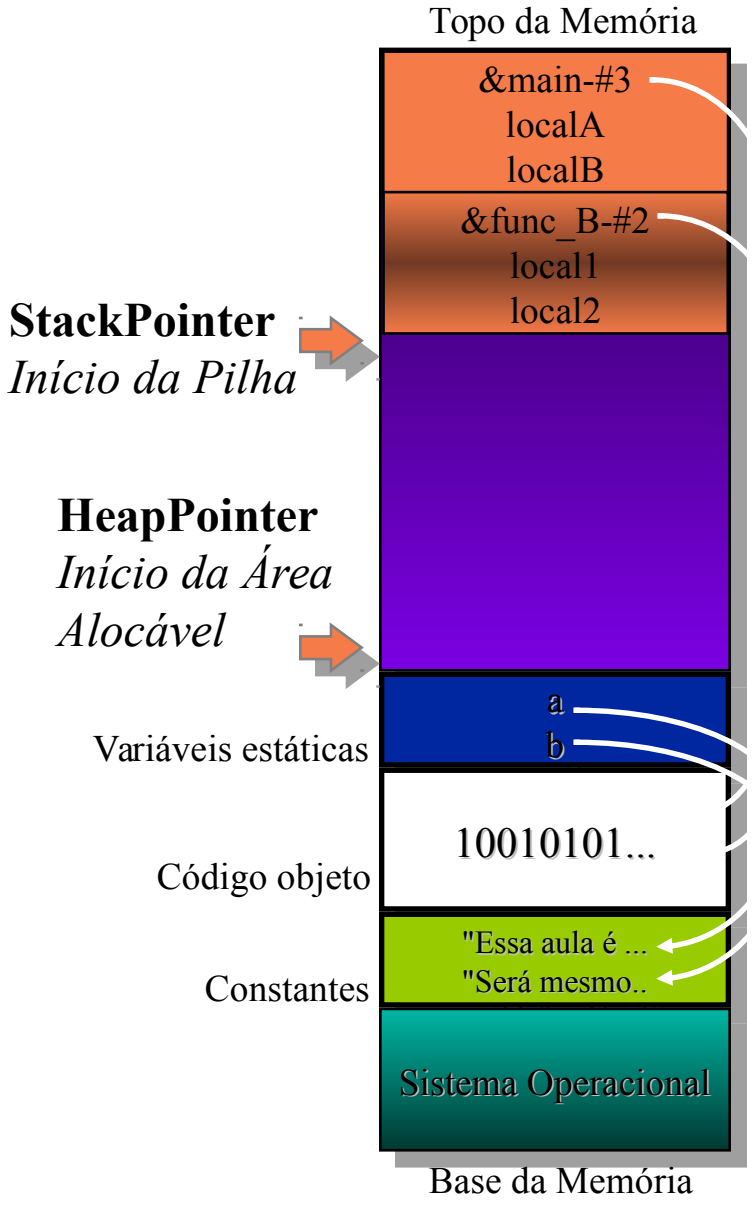


```
#include <stdio.h>
char *a, *b;

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();
}
```



```
#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();
}
```

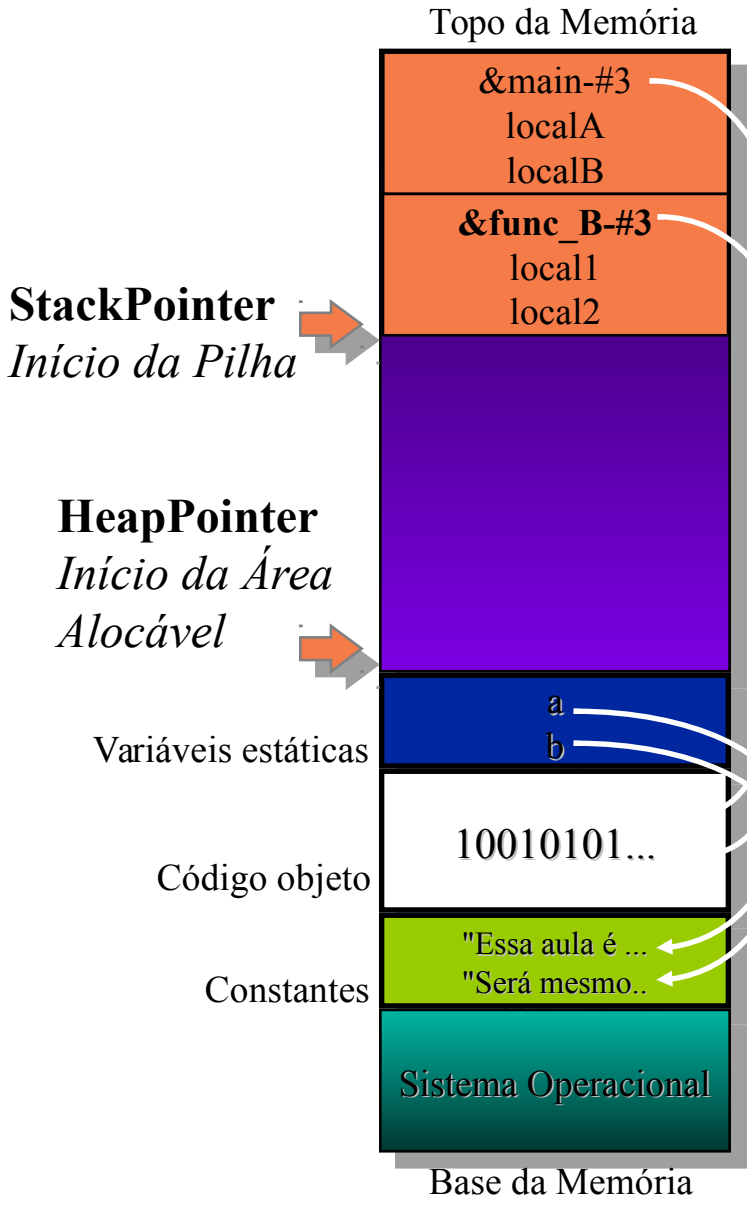


```
#include <stdio.h>
char *a, *b;

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();
}
```

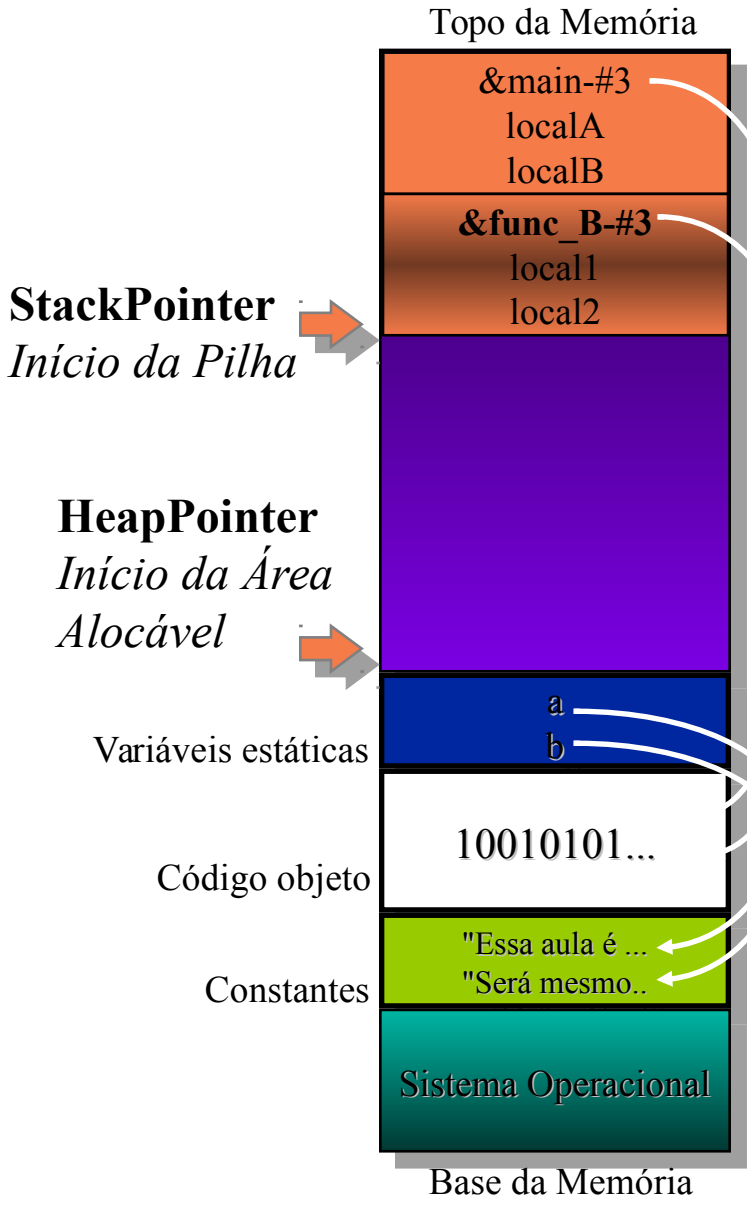


```
#include <stdio.h>
char *a, *b;

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();
}
```



```
#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();
}
```



```
#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();
}
```



creative commons
COMMONS DEED

Atribuição-Uso Não-Comercial-Compartilhamento pela Licença 2.5 Brasil

Você pode:
- copiar, distribuir, exibir e executar a obra
- criar obras derivadas

Sob as seguintes condições:
Atribuição — Você deve dar crédito ao autor original, da forma especificada pelo autor ou licenciante.
Uso Não-Comercial — Você não pode utilizar esta obra com finalidades comerciais.
Compartilhamento pela mesma Licença — Se você alterar, transformar, ou criar outra obra com base nesta, você somente poderá distribuir a obra resultante sob uma licença idêntica a esta.

Para ver uma cópia desta licença, visite http://creativecommons.org/licenses/by-nc-sa/2.5/br/ ou mande uma carta para Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.