Introdução à Computação II

Apontadores e alocação de memória

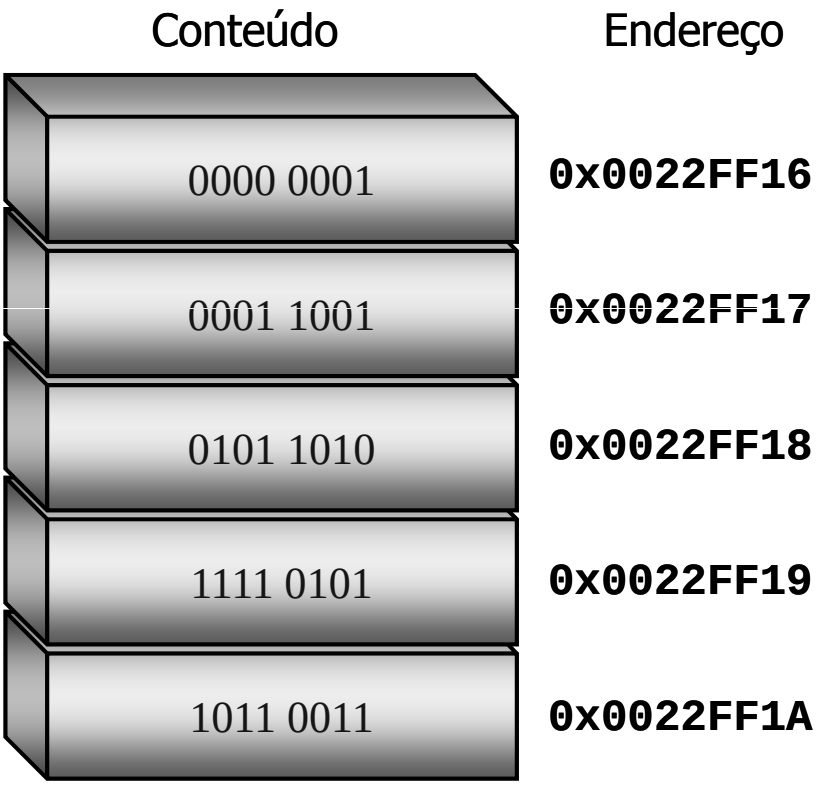
Profa.: *Adriane Beatriz de Souza Serapião*

adriane@rc.unesp.br

Endereços

- A memória de qualquer computador é uma sequência de bytes.
- Cada byte pode armazenar um número inteiro entre 0 e 255.
- Cada byte na memória é identificado por um endereço numérico, independente do seu conteúdo.

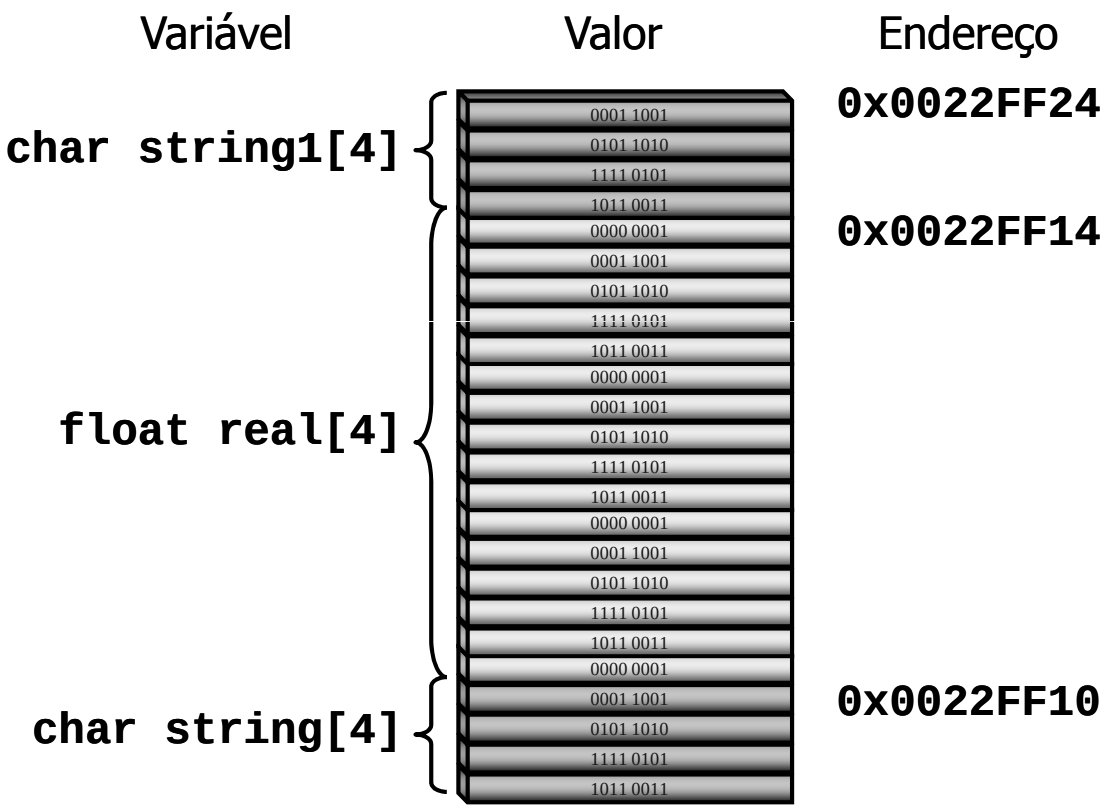
Endereços



Endereços

- **Cada objeto (variáveis, strings, vetores, etc.) que reside na memória do computador ocupa um certo número de bytes:**
  - **Inteiros:** **4 bytes consecutivos**
  - **Caracteres:** **1 byte**
  - **Ponto-flutuante:** **4 bytes consecutivos**
- **Cada objeto tem um endereço.**

Endereços



Endereços - resumo

```
int x = 100;
```

- **Ao declararmos uma variável x como acima, temos associados a ela os seguintes elementos:**
  - **Um nome (x)**
  - **Um endereço de memória ou referência (0xbfd267c4)**
  - **Um valor (100)**
- **Para acessarmos o endereço de uma variável, utilizamos o operador &**

As Funções de Alocação Dinâmica de Memória em C

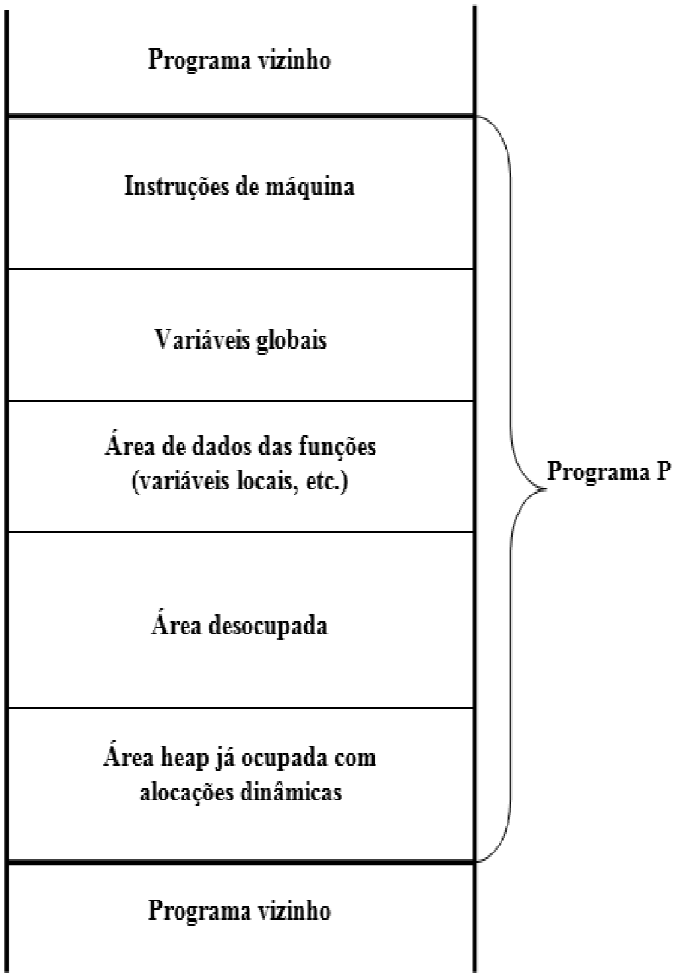
- ***Alocação Dinâmica*** é um meio pelo qual o programa pode obter memória enquanto está em execução.

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*).**

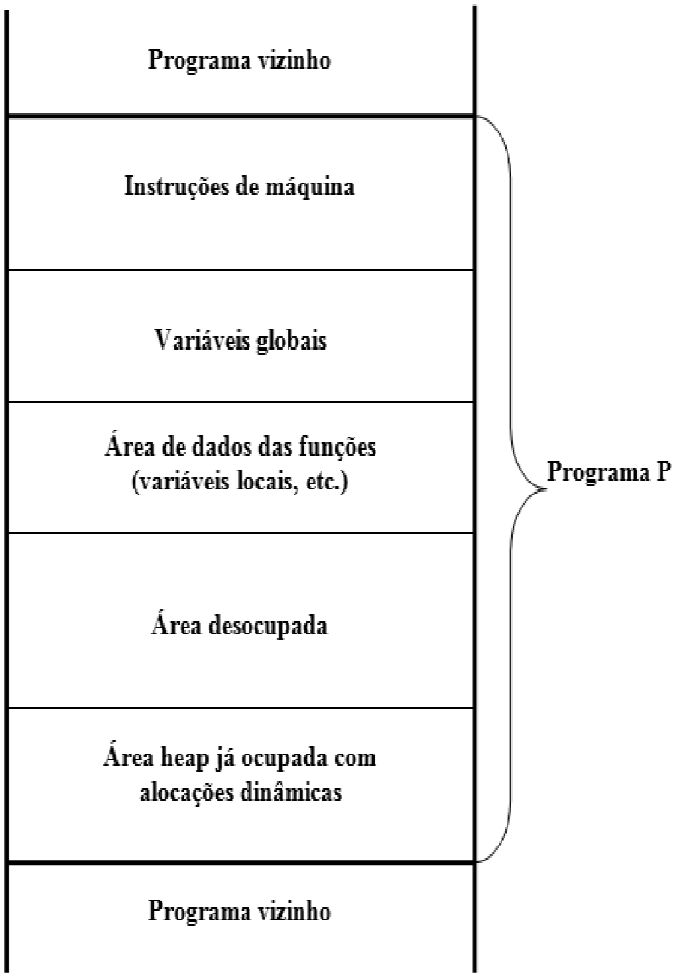
Alocação de memória

- **O layout da área ocupada pelo programa é planejado pelo compilador**
- **O gerenciamento durante a execução fica por conta do próprio programa**
- **A área total reservada para o programa é fixa**
- **As regiões das instruções e das variáveis globais também tem tamanho fixo**



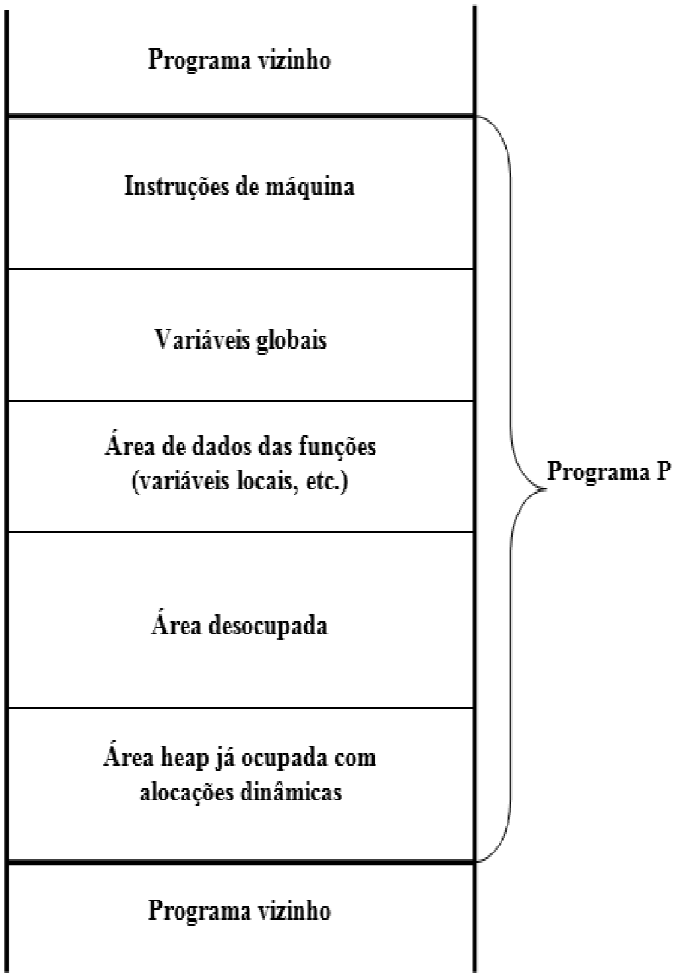
Alocação de memória

- As outras regiões tem tamanho variável durante a execução
- A área de dados das funções destina-se a guardar os parâmetros, as variáveis locais e informações operacionais das versões ativas de funções num dado momento
- Essa área varia de tamanho, pois essas versões de funções não ficam ativas o tempo todo



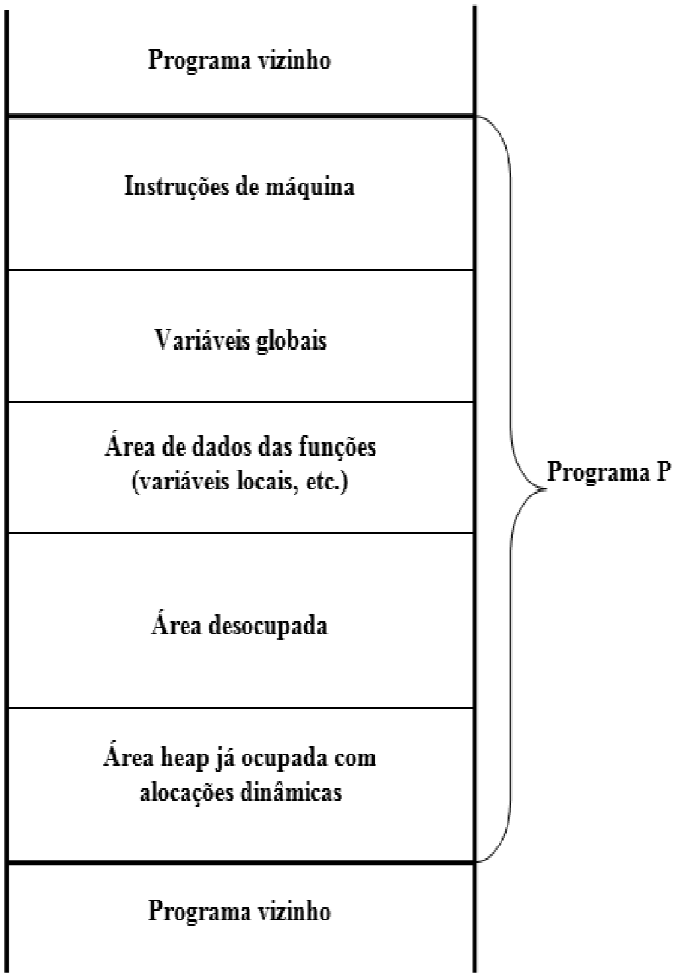
Alocação de memória

- Quando uma versão de função é chamada para execução, sua área de dados é carregada na memória
- O carregamento é feito a partir da fronteira com a área desocupada
- Quando sua execução é encerrada, sua área é retirada da memória, aumentando a área desocupada



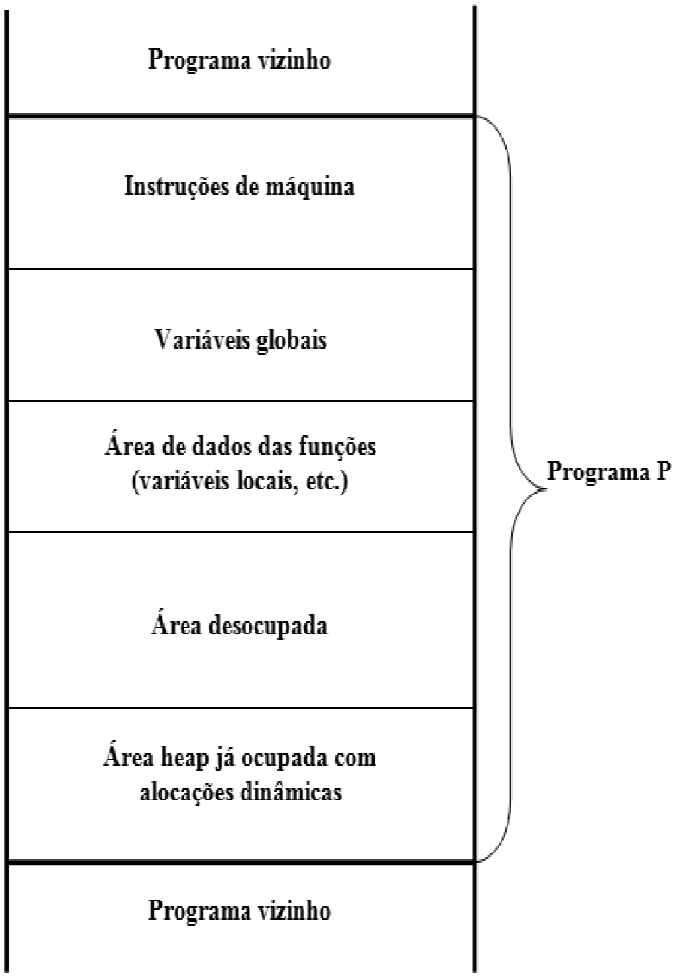
Alocação de memória

- A área heap aumenta a partir de sua fronteira com a área desocupada
- Isso acontece quando uma alocação dinâmica de memória é feita (malloc ou outras do gênero)
- A área de dados das funções aumenta para baixo e a heap aumenta para cima



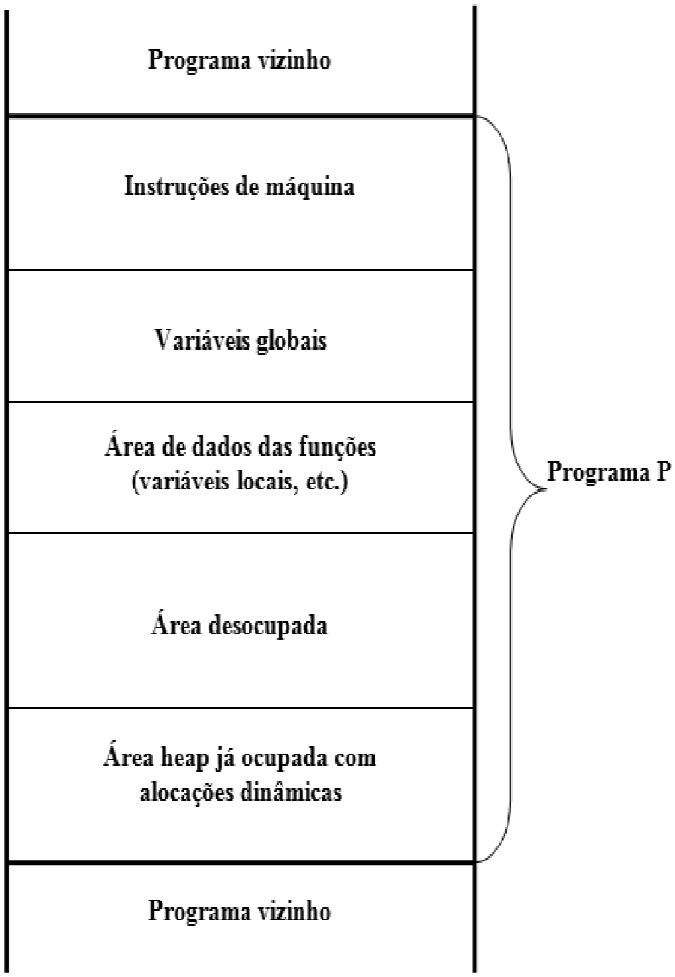
Alocação de memória

- Nesse processo, se o programador não tomar cuidado, essas duas áreas podem se encontrar
- Aí, esgota-se a capacidade da área desocupada
- Novas chamadas de funções e novas alocações dinâmicas ficam impossibilitadas



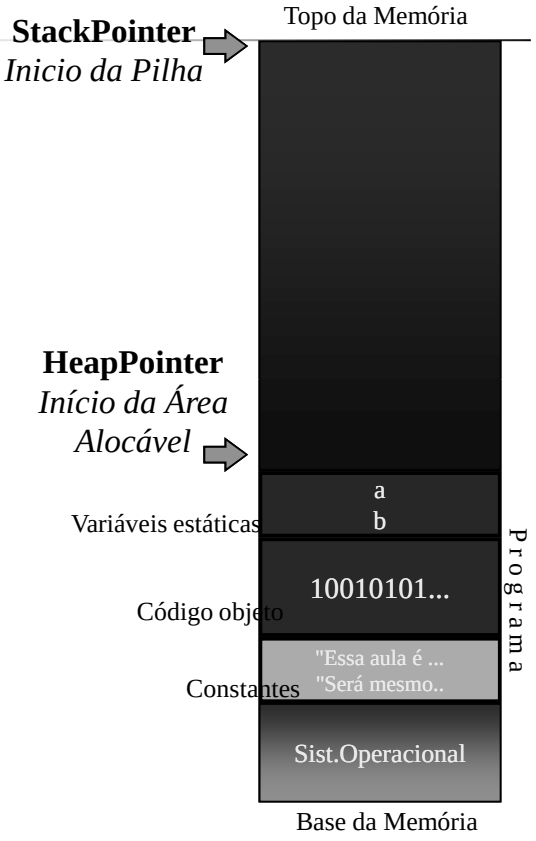
Alocação de memória

- A função free torna a deixar disponível a área reservada numa alocação dinâmica
- Seu parâmetro é um ponteiro
- Ela re-disponibiliza a área previamente alocada e apontada por ele
- Deve-se usar essa função toda vez que uma alocação não tiver mais utilidade para o programa



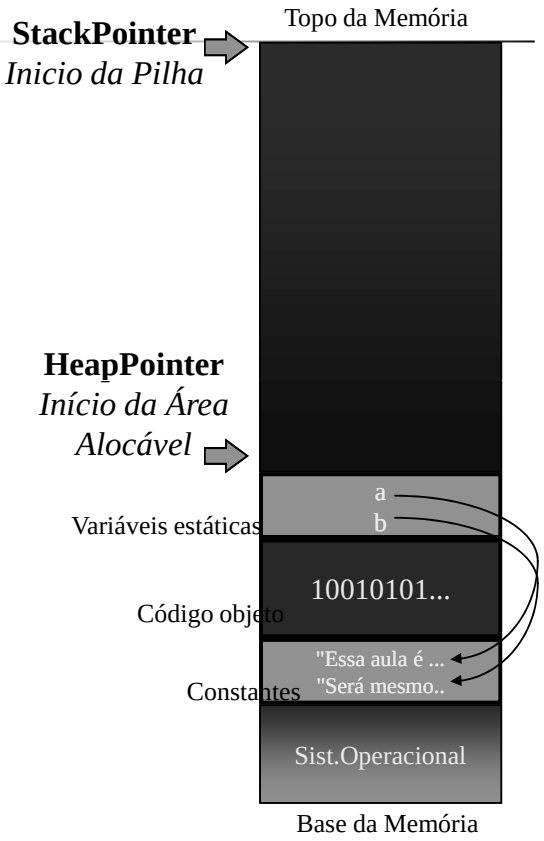
- **Programa:**

```
#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();
}
```



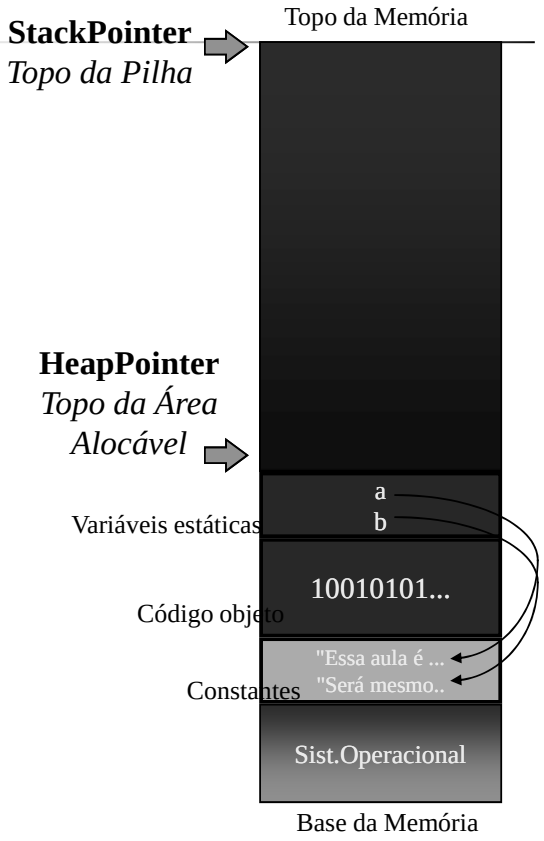
Programa:

```
#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();
}
```



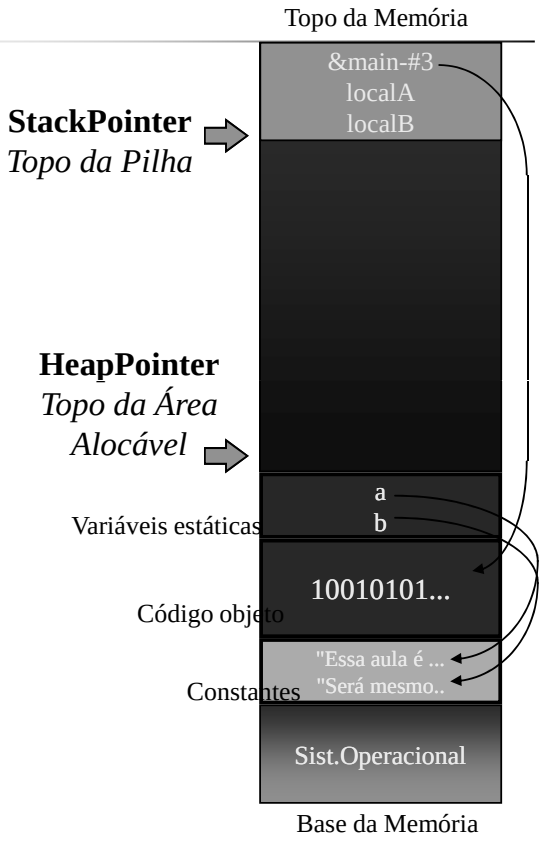
Programa:

```
#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();
}
```



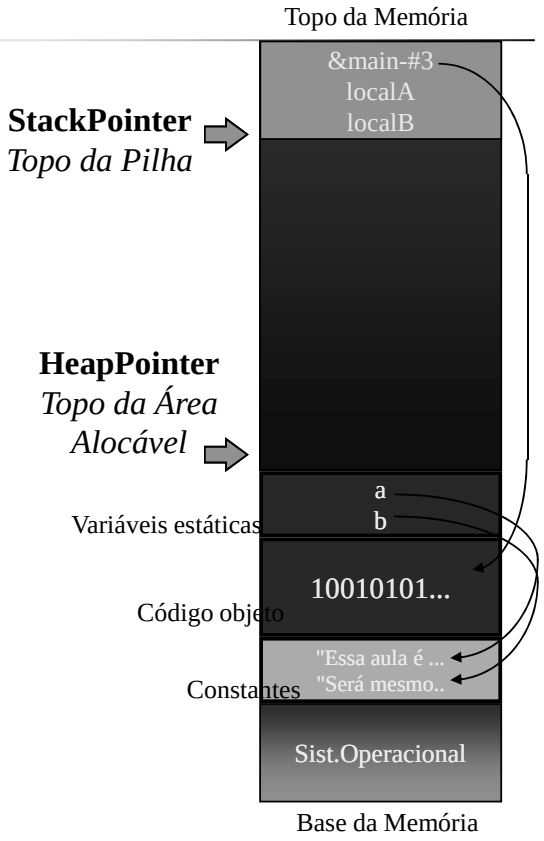
Programa:

```
#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();
}
```



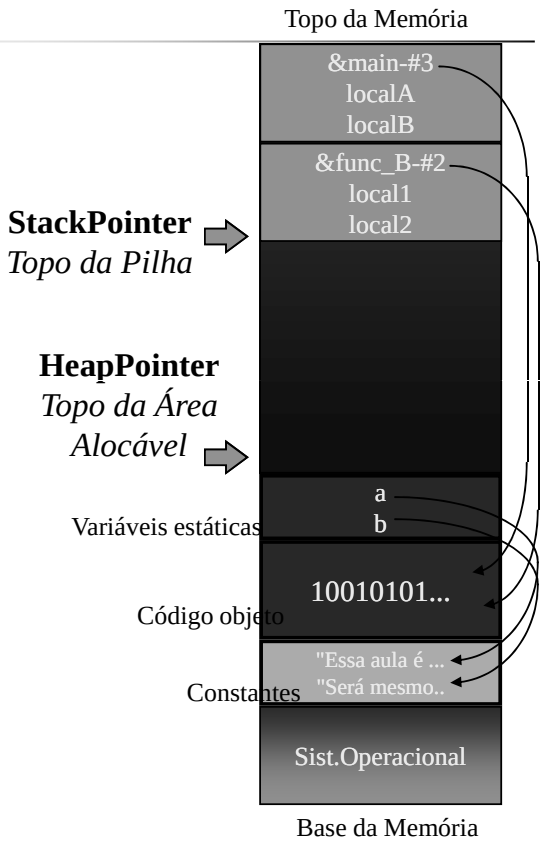
Programa:

```
#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();
}
```



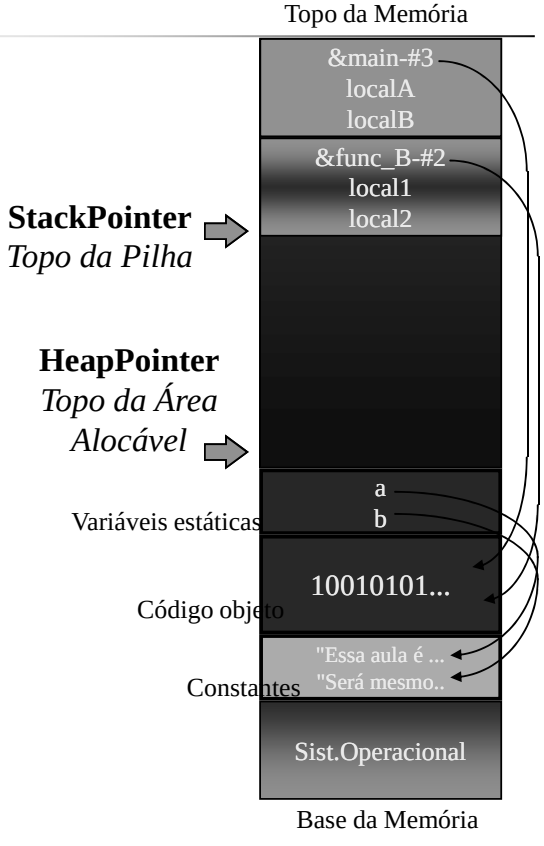
Programa:

```
#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();
}
```



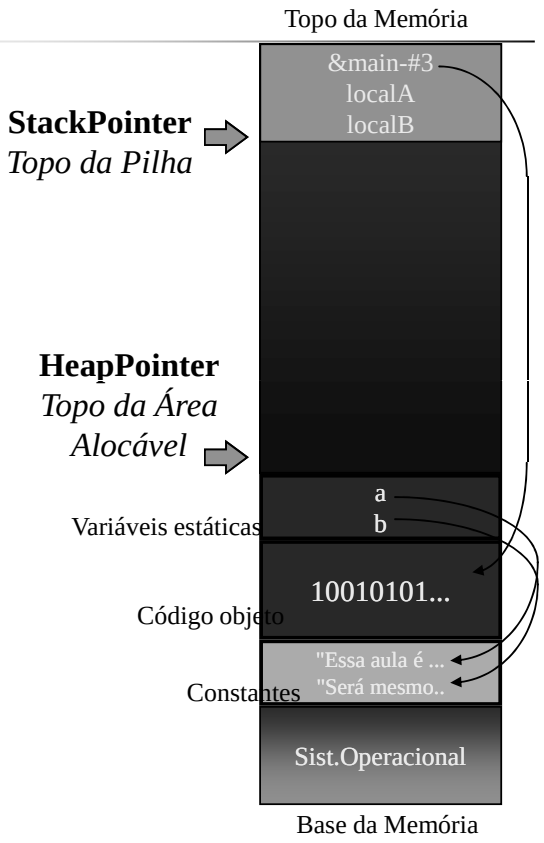
Programa:

```
#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();
}
```



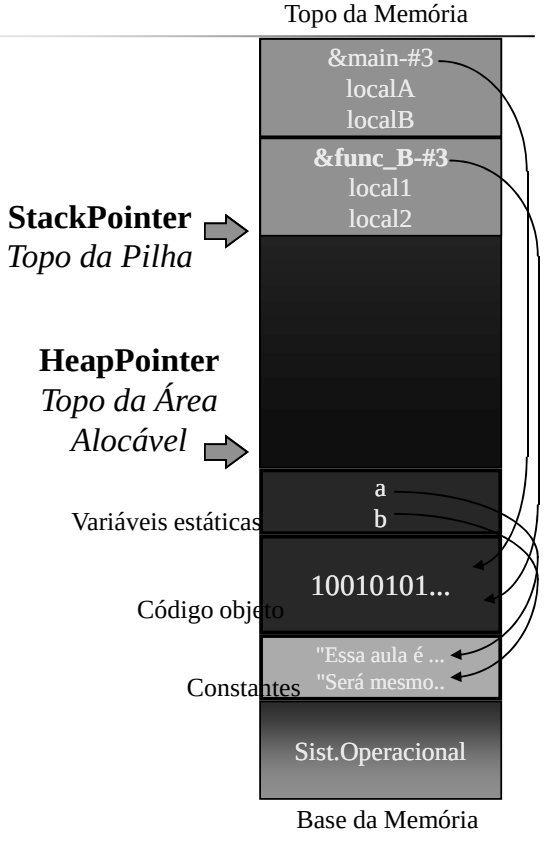
Programa:

```
#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();
}
```



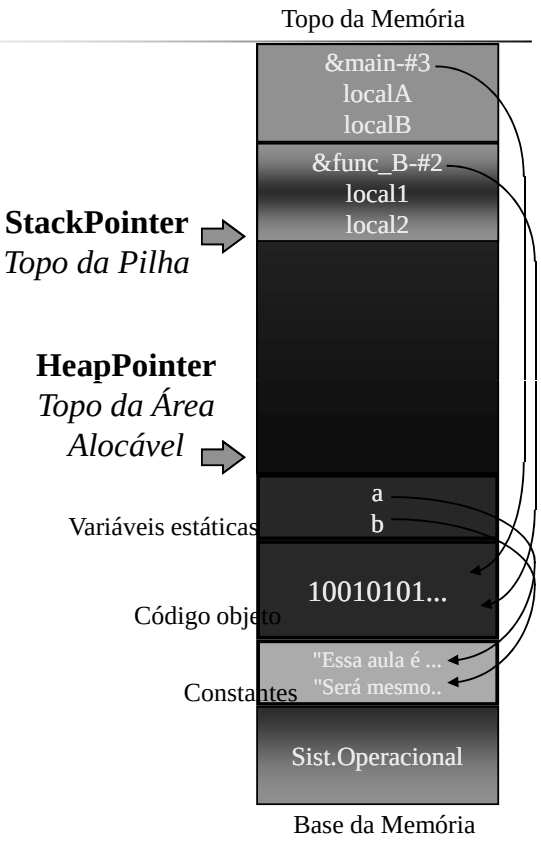
Programa:

```
#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();
}
```



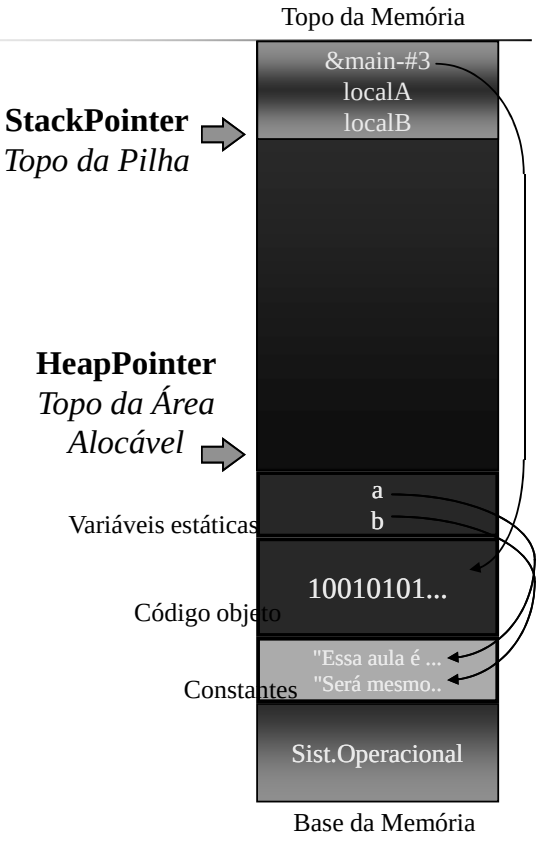
Programa:

```
#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();
}
```



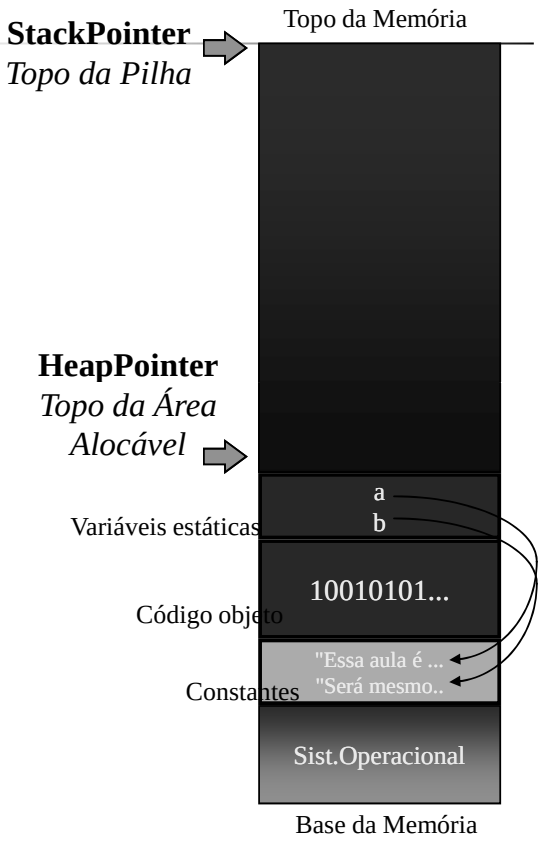
Programa:

```
#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();
}
```



- **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();
}
```



Introdução

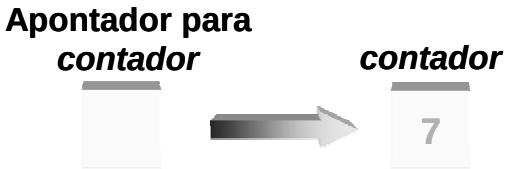
- **Apontadores**
  - **Poderosos, apesar de difícil domínio.**
  - **Simulação de *chamadas por referência.***
  - **Relação íntima com *arrays* e cadeias de caracteres (*strings*).**

Declaração e Inicialização de Variáveis para Apontamento

- Variáveis para Apontamento
  - Contêm endereços de memória como valores.
  - Variáveis normais contêm valores específicos (referência direta).

Declaração e Inicialização de Variáveis para Apontamento

- Variáveis para Apontamento
  - Apontadores contêm endereços de variáveis que contêm valores específicos (referência indireta).
  - Referência indireta ⇨ referenciação de um valor via apontador.



Declaração e Inicialização de Variáveis para Apontamento

- **Declaração de Apontadores**
  - *** ⇨ Usado com variáveis de apontamento:**

    int *apont;
  - **Declaração de um apontador para um *int* (apontador do tipo *int **).**
  - **Apontadores múltiplos requerem o uso de um * antes de cada declaração de variável:**

    int *apont1, *apont2;

Declaração e Inicialização de Variáveis para Apontamento

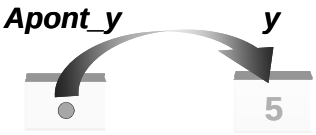
- **Declaração de Apontadores**
  - **Possibilidade de declaração de apontadores qualquer tipo de dados.**
  - **Inicialização de apontadores para *0*, *NULL* ou um endereço.**
    - ***0* ou *NULL* ⇨ apontadores para nada (*NULL* preferencialmente).**

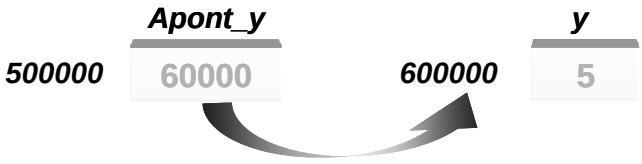
Operadores de apontamento

- *&* (Operador de endereçamento)
  - Retorna o endereço de um operando:

```
int y = 5;
int *apont_y;
apont_y = &y;   // apont_y contém o endereço de y
```

apont_y* "aponta para" *y





O endereço de *y* é o valor de *apont_y*

Operadores de apontamento

- * (Operador de referência indireta/ desreferência)
  - Retorna uma cópia do conteúdo da locação para a qual o operando aponta.
  - ***apont_y*** retorna ***y*** (visto que ***apont_y*** aponta para ***y***):
  - * pode ser usado para atribuição.
    - Retorna cópia para um objeto:

```
*apont_y = 7;   // altera o valor de y para 7
```

Operadores de apontamento

- *** (Operador de referência indireta ou de desreferência).**
  - **Operando de * deve ser um *lvalue* (não constante).**
- *** e & são inversos.**
  - **Um cancela o outro.**

Operadores de apontamento

```
01  /* Uso dos operadores & e * */
02
03  #include <stdio.h>
04
05  int main()
06  {
07     int a;       /* a é um inteiro */
08     int *aPont;  /* aPont é um apontador para um inteiro */
09
10     a = 7;
11     aPont = &a;  /* aPont aponta para o endereço de a */
12
13     printf( "O endereço de a é %p "\nO valor de aPont é %p ", &a, aPont );
14
15     printf( "\n\nO valor de a é %d " "\ nO valor de *aPont é %d ", a, *aPont );
16
17     printf( "\n\n Prova de que * e & são complementares\n&*aPont = %p "
18             "\n*&aPont = %p\n ", &*aPont, *&aPont )
19
20     return 0;
21  }
```

O endereço de **a** é o valor de **aPtr**

O operador * retorna uma cópia para a locação para a qual seu operando aponta. **aPtr** aponta para **a**, de modo que ***aPtr** retorna **a**.

Observar que * e & são inversos

Operadores de apontamento

- **Saída apresentada**

```
O endereço de a é 0012FF88
O valor de aPont é 0012FF88

O valor de a é 7
O valor de *aPont é 7
Prova de que * e & são complementares
&*aPont = 0012FF88
*&aPont = 0012FF88
```


Chamadas de funções por referência

- **Chamada por referência com argumentos de apontamento:**
  - **Passagem do endereço do argumento via operador &.**
  - **Possibilidade de alteração da locação corrente na memória.**
  - **Impossibilidade de passagem de *arrays* com & (o nome de um *array* já é um apontador).**

Chamadas de funções por referência

- **Operador ***
  - **Uso como cópia/ "apelido" da variável interna à função:**

```
void double(int *num)
{
 *num = 2*(*num);
}
```

  - **Uso de **num* como "apelido" da variável passada.**

Chamadas de funções por referência

```
01 /* Uso de chamada por referência para cálculo do cubo de uma
   variável com um argumento apontador */
02
03 #include <stdio.h>
04
05 void cuboPorReferencia( int * );   /* protótipo */
06
07 int main()
08 {
09    int num = 5;
10
11    printf( "O valor original de num eh %d", num );
12    cuboPorReferencia( &num );
13    printf( "\nO novo valor de  num eh %d\n", num );
14
15    return 0;
16 }
17
18 void cuboPorReferencia( int *nApont
19 {
20    *nApont = *nApont * *nApont * *nApont;  /* cubo em main */
21 }
```

*O protótipo da função inclui um apontador para um inteiro (**int ***)*

*O endereço de um número é dado - **cuboPorReferencia** espera um apontador (o endereço de uma variável)*

****nApont** é usado em **cuboPorReferencia** (***nApont** é um número)*

Uso do Qualificador *const* com Apontadores

- Qualificador *const*
  - Variável não pode ser alterada.
  - Uso de *const* se a função não precisar alterar uma variável.
  - Tentativa de alteração de uma variável *const* produz um erro.

Uso do Qualificador *const* com Apontadores

- Apontadores *const*
  - Apontamento para uma locação de memória constante.
  - Inicialização obrigatória no ato da declaração.
  - *int *const Apont = &x;*
    - Tipo *int *const* ⇨ Apontador constante para um *int.*
  - *const int * Apont = &x;*
    - Apontador regular para um *const int*.

Uso do Qualificador *const* com Apontadores

- **Apontadores *const***
  - ***const int *const Apont = &x;***
    - **Apontador *const* para um *const int.***
    - **Possibilidade de alteração de *x*, mas não de ** Apont.***

Uso do Qualificador *const* com Apontadores

```
01 /*Tentativa de modificação de um apontador constante para dados não
02 constantes*/
03
04 #include <stdio.h>
05
06 int main()
07 {
08    int x, y;
09    int *const apont = &x; /* apont é um apontador constante para um
10    inteiro, passível de modificação através de apont, embora este
11    aponte sempre para a mesma locação de memória. */
12    *apont = 7;
13    apont = &y;
14    return 0;
15 }
```

*A alteração de ***apont** é correto, uma vez que **x** não é constante*

*A alteração de **apont** é um erro, uma vez que se trata de um apontador constante*

```
Error E2024 FIG07_13.c 16: Cannot modify a const object in function
main
*** 1 errors in Compile ***
```

Ordenação Borbulhante (*Bubble Sort*) Usando Chamada por Referência

- **Implementação da ordenação borbulhante usando apontadores.**
  - **Varredura de dois elementos.**
  - **Passagem de endereços (uso de *&*) de elementos do *array* para a função *swap*.**
    - **Array elements have call-by-value default**
  - **Uso de apontadores e do operador *** **para o chaveamento de elementos do *array* pela função *swap*.**

Ordenação Borbulhante (*Bubble Sort*) Usando Chamada por Referência

- **Pseudo-código**

Inicializar array
Imprimir os dados na ordem original
Chamar a função de ordenação borbulhante (bubblesort)
Imprimir o array produzido pela ação da função
Definir ordenação borbulhante (bubblesort)

Ordenação Borbulhante (*Bubble Sort*) Usando Chamada por Referência

- ***sizeof***
  - Retorno do tamanho do operando (em Bytes).
  - Para *arrays* ⇨ tamanho de 1 elemento * número de elementos.
  - Se ***sizeof(int)*** igual a ***4*** bytes, então:

```
int meuArray[ 10 ];
printf( "%d", sizeof( meuArray ) );
```

    - imprimirá ***40***.

Ordenação Borbulhante (*Bubble Sort*) Usando Chamada por Referência

- Possibilidade de uso de ***sizeof*** com:
  - Nomes de variáveis
  - Nome de tipos
  - Valores constantes

Ordenação Borbulhante (*Bubble Sort*) Usando Chamada por Referência

```
01  /*Programa que organiza valores de um array em ordem crescente, imprimindo o
02  resultado*/
03  #include <stdio.h>
04  #define TAMANHO 10
05  void borbulha( int *, const int );
06
07  int main()
08  {
09     int a[TAMANHO] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
10     int i;
11     printf( "Dados na ordem original \n" );
12     for ( i = 0; i < TAMANHO ; i++ )
13        printf( "%4d", a[ i ] );
14     borbulha( a, TAMANHO );        /* ordena e apresenta o array */
15     printf( "\nDados em ordem crescente\n" );
16     for ( i = 0; i < TAMANHO ; i++ )
17        printf( "%4d", a[ i ] );
18     printf( "\n" );
19     return 0;
20  }
```

borbulha *recebe endereços de elementos do array (uso de apontadores). O nome de um array é um apontador.*

Ordenação Borbulhante (*Bubble Sort*) Usando Chamada por Referência

```
21  /*Função que varre um array, organizando seus elementos em ordem crescente
22  void borbulha( int *array, const int tam )
23  {
24     void permuta( int *, int * );
25     int i, j;
26     for ( i = 0; i < tam - 1; i++ )
27        for ( j = 0; j < tam - 1; j++ )
28           if ( array[ j ] > array[ j + 1 ] )
29              permuta( &array[ j ], &array[ j + 1 ] );
30  }
31  void permuta( int *apontelem1, int *apontelem2)
32  {
33     int auxi = *apontelem1;
34     *apontelem1 = *apontelem2;
35     *apontelem2 = auxi;
36  }
```

permuta *é chamada por* ***borbulha*** *para fazer a permuta de elementos, no processo de ordenação, quando necessária*

```
Dados na ordem original
   2   6   4   8  10  12  89  68  45  37
Dados em ordem crescente
   2   4   6   8  10  12  37  45  68  98
```

Expressões e Aritmética de Apontadores

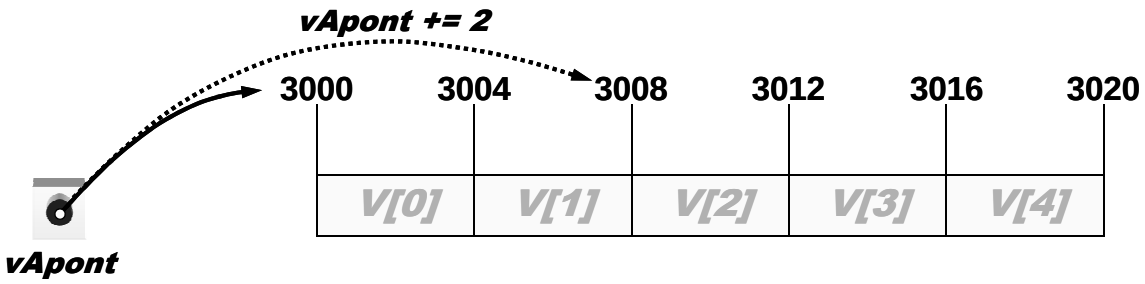
- **Operações Aritméticas sobre Apontadores**
  - **Incremento/Decremento (*++*/*--*).**
  - **Adição de um inteiro a um apontador(*+* ou *+=* , *-* ou *-=*).**
  - **Subtração de Apontadores.**
  - **Operações sem sentido, a menos que executadas sobre um *array.***

Expressões e Aritmética de Apontadores

- ***Array int* com 5 elementos com *int* armazenado em 4 bytes de memória.**
  - ***vApont* aponta para o primeiro elemento *v[0]***
    - ***vApont = 3000* (armazenado na locação *3000*)**
  - ***vApont += 2;* redireciona *vApont* para *3008***
    - ***vApont* passará a apontar para *v[2]* (incremento de 2)**

Expressões e Aritmética de Apontadores

- ***Array int*** **com 5 elementos com** ***int*** **armazenado em 4 bytes de memória.**



Expressões e Aritmética de Apontadores

- **Subtração de Apontadores**
  - **Retorno do número de elementos de um para o outro.**
    - **Exemplo:**

      vApont2 = v[2];
      vApont1 = v[0];

    - ***vApont2 - vApont1*** **produzirá** ***2*** **como resultado.**

Expressões e Aritmética de Apontadores

- **Comparação de Apontadores (*<*, *==* , *>*)**
  - **Só há sentido se ambos os apontadores são relativos ao mesmo *array.***
  - **Exemplo:**
    - **Verificação de qual dos apontadores aponta para o elemento de maior índice do *array.***
    - **Verificação para determinar se se trata de um apontador *NULL*.**

Expressões e Aritmética de Apontadores

- **Apontadores do mesmo tipo podem ser atribuídos um ao outro.**
  - **Tipos diferentes ⇨ Necessidade de um operador de conversão.**
    - **Transformação do tipo do apontador à direita da atribuição para o tipo do operador à esquerda da atribuição.**

Expressões e Aritmética de Apontadores

- **Apontadores do mesmo tipo podem ser atribuídos um ao outro.**
  - **Exceção ⇨ Apontador para *void* (tipo *void **).**
    - **Apontador genérico ⇨ Representação de qualquer tipo de apontador.**
    - **Todos os tipos de apontadores podem ser atribuídos a um apontador para *void*.**
    - **Nenhuma conversão é necessária.**
    - **Apontadores para *void* não podem ser desreferenciados.**

Relação entre Apontadores e Arrays

- ***Arrays* e Apontadores**
  - **Relação íntima ⇨ Uso quase indiferente de ambos.**
  - **Nome de um *array* ⇨ Apontador constante.**
  - **Possibilidade de execução de operações de subscrição a partir do uso de apontadores.**
    - **Exemplo ⇨ Declaração de um *array* *b[5]* e um apontador *bApont*.**
    - **Para o uso indistinto de ambos:**
      - ***bApont = b;* ⇨ *bApont = &b[0];***
        - **Atribuição explícita de *bApont* para endereçamento do primeiro elemento de *b*.**

Relação entre Apontadores e Arrays

- ***Arrays* e Apontadores**
  - **Possibilidade de execução de operações de subscrição a partir do uso de apontadores.**
    - **Elemento *b[3]***
      - **Possibilidade de acesso a partir de **(bApont + 3)*:**
        - ***3*** é o *offset.*
        - ****(bApont + n)*** ⇨ Notação *apontador/offset.*
      - **Possibilidade de acesso a partir de *bApont[3]*:**
        - Notação *apontador/subscrito;*
          - ***bApont[3] igual a b[3].***
      - **Possibilidade de acesso a partir da aritmética de apontadores:**
        - ****(b + 3)***

Arrays de apontadores

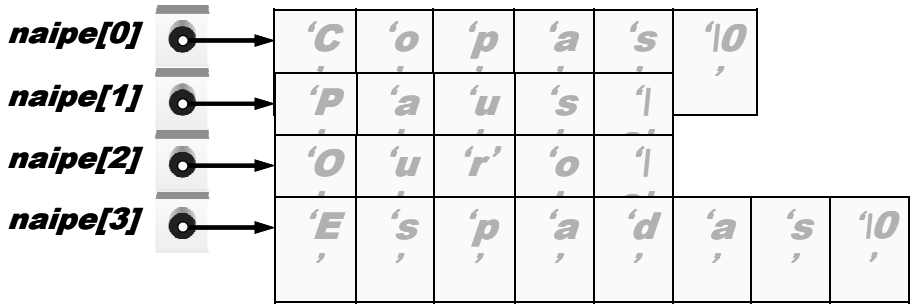
- ***Arrays* podem conter apontadores**
  - **Exemplo**
  - ***Array* de cadeias de caracteres:**

    char *naipe[4]={“Copas", “Paus", “Ouro", “Espadas" };

    - **Cadeias de caracteres são apontadores para o primeiro caractere.**
    - ***char *** ⇨ **Cada elemento de *naipe* é um apontador para um *char*.**
    - **Cadeias de caracteres não são realmente armazenadas no *array naipe*, apenas os apontadores para as cadeias de caracteres o são.**

Arrays de apontadores

- Exemplo
  - O *array* ***naipe*** tem tamanho fixo.
  - As cadeias de caracteres podem ter qualquer tamanho.



Arrays de apontadores

- Ponteiros e matrizes:
  - No programa abaixo entenda que <u>o nome de uma matriz sem índice é o endereço do início da matriz.</u>
- Exemplo relacionamento entre ponteiros e matrizes:

```
char str[80], *p;
char *p1;
p1 = str;
```

- Pode-se colocar `str[4]` ou `*(p1+4);`
- A aritmética de ponteiros pode ser mais rápida do que a indexação de matrizes!!!

Arrays de apontadores

Exemplo indexação com matrizes

```
main()
{
   char str[80];
   int i;
   printf("digite uma string em minúsculas: ");
   gets (str);
   printf ("transformei para: );
   for (i=0;str[i];i++)
   printf ("%c",tolower(str[i]);
}
```

Exemplo acesso com ponteiros

```
main()
{
   char str[80], *p;
   printf("digite uma string em minúsculas: ");
   gets (str);
   printf ("transformei para: );
   p = str;
   while (*p) printf ("%c",tolower(*p++))
}
```

– Acesso aleatório é melhor a indexação

Arrays de apontadores

- **Indexando um ponteiro – pode se indexar um ponteiro como se ele fosse uma matriz.**

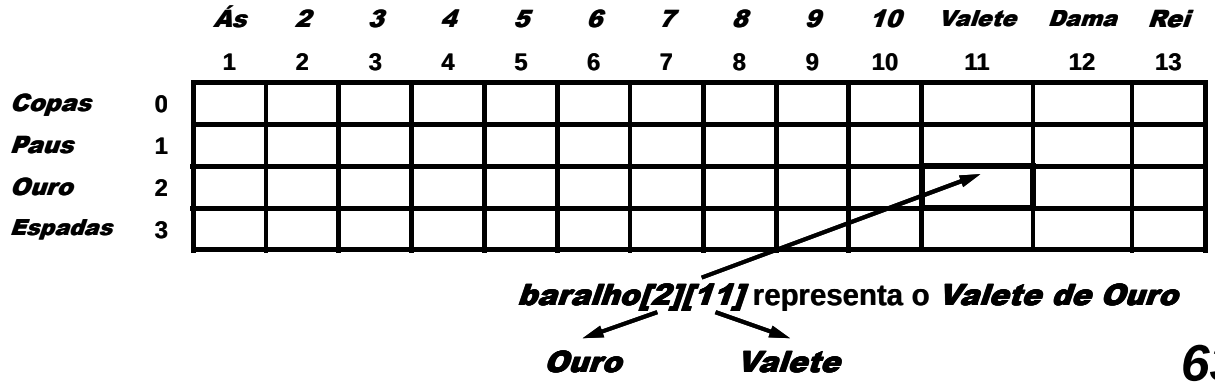
Exemplo indexação de ponteiros

```
main()
{
   int i[5]={1,2,3,4,5}
   int *p, t;
   p = i;
   for (t=0;t<5;t++) printf("%d",p[t]);
}
```

Em C, o comando p[t] é o mesmo que *(p+t)

Estudo de Caso: Simulação de Embaralhamento e Distribuição de Cartas

- Programa simulador do embaralhamento de cartas:
  - Uso de um *array* de apontadores para cadeias de caracteres.
  - Uso de um *array* bidimensional (naipe, face).



Estudo de Caso: Simulação de Embaralhamento e Distribuição de Cartas

- Programa simulador do embaralhamento de cartas:
  - Inclusão dos números 1-52 ao *array.*
    - Representação da ordem na qual as cartas são distribuídas.

Estudo de Caso: Simulação de Embaralhamento e Distribuição de Cartas

- **Pseudo-código**
  - **Nível de Topo (*Top level*)**

    Embaralhar e distribuir 52 cartas

  - **Primeiro refinamento**

    Inicializar o array naipe
    Inicializar o array face
    Inicializar o array baralho
    Embaralhar o baralho
    Distribuir 52 cartas

Estudo de Caso: Simulação de Embaralhamento e Distribuição de Cartas

- **Segundo refinamento**
  - **Conversão de *Embaralhar e distribuir 52 cartas* para:**

    Para cada uma das 52 cartas
    Colocar o número da carta em um espaço desocupado do baralho, selecionado aleatoriamente

  - **Conversão de *Distribuir 52 cartas* para:**

    Para cada uma das 52 cartas
    Determinar o número da carta no baralho e imprimir o valor da face e o naipe da carta

Estudo de Caso: Simulação de Embaralhamento e Distribuição de Cartas

- **Terceiro refinamento**
  - **Conversão de *Embaralhar e distribuir 52 cartas* para:**

Escolher aleatoriamente um espaço no baralho
Ao escolher um espaço já escolhido anteriormente
Escolher aleatoriamente um espaço no baralho
Colocar o número da carta no espaço escolhido do baralho

Estudo de Caso: Simulação de Embaralhamento e Distribuição de Cartas

- **Terceiro refinamento**
  - **Conversão de *Distribuir 52 cartas* para:**

Para cada espaço no array do baralho
Se o espaço tiver um número de carta
Imprimir o valor da face e o naipe da carta

Estudo de Caso: Simulação de Embaralhamento e Distribuição de Cartas

Programa Principal

```
01  /* Programa para Embaralhamento e Distribuição de Cartas*/
02  #include <stdio.h>
03  #include <stdlib.h>
04  #include <time.h>
05
06  void embaralha(int [][ 13 ]);
07  void distribui(const int [][ 13 ], const char *[], const char *[]);
08
09  int main()
10  {
11     const char *naipe[ 4 ] = {“Copas", “Paus", “Ouro", “Espadas"};
12     const char *face[ 13 ] = {“Ás", "Dois", "Três", “Quatro", “Cinco”, “Seis", "Sete",
13                               “Oito", “Nove", “Dez”, “Valete", “Dama", “Rei"};
14     int baralho[ 4 ][ 13 ] = { 0 };
15
16     srand( time( 0 ) );
22     embaralha(baralho);
23     distribui(baralho, face, naipe);
25     return 0;
26  }
```


Estudo de Caso: Simulação de Embaralhamento e Distribuição de Cartas

```
27  void embaralha(int cartas[][13])
28  {
29    int linha, coluna, carta;
30
31    for (carta = 1; carta <= 52; carta ++){
32      do {
33          linha = rand()%4;
34          coluna = rand()%13;
35      }while(cartas[linha][coluna] != 0);
36      cartas[linha][coluna] = carta;
37    }
38  }
39  void distribui( const int cartas[][ 13 ], const char *faces[], const char *naipes[] )
40  {
41    int carta, linha, coluna;
42    for (carta = 1; carta <= 52; carta++)
43      for (linha = 0; linha <= 3; linha ++)
44        for (coluna = 0; coluna <= 12; coluna++)
45          if (cartas[linha][coluna] == carta)
46            printf( "%5s de %-8s%c", faces[coluna], naipes[linha], carta%2 == 0 ? '\n' : '\t' );
47  }
```

*Os números **1-52** são aleatoriamente inseridos no array de cartas.*

Busca pelo número da carta no baralho, imprimindo o valor da face e naipe.

Estudo de Caso: Simulação de Embaralhamento e Distribuição de Cartas

- **Exemplo de Saída**

```
Seis de Paus
Ás de Espadas
Ás de Copas
Dama de Paus
Dez de Copas
Dez de Espadas
Dez de Ouro
Quatro de Ouro
Seis de Ouro
Oito de Copas
Nove de Copas
Dois de Espadas
Cinco de Paus
Dois de Ouro
Cinco de Espadas
Rei de Ouro
Dois de Copas
.
.
.
```


Apontadores para Funções

- **Apontador para função:**
  - **Contém endereço da função na memória.**
  - **Similar ao fato do nome do *array* ser o endereço do primeiro elemento.**
  - **Nome da função é o endereço inicial do código que realiza a tarefa da função.**
- **Possibilidades de apontadores para funções:**
  - **Passagem para funções.**
  - **Armazenamento em *arrays.***
  - **Atribuição a outros apontadores para função.**

Apontadores para Funções

- **Exemplo: Ordenação borbulhante (*Bubble Sort*)**
  - **Função *borbulha* usa um apontador para função:**
    - ***borbulha* chama uma função auxiliar (*permuta*).**
    - ***permuta* determina a ordenação ascendente ou descendente.**
  - **Argumento em *borbulha* para o apontador para *permuta*:**

    bool (*compare)(int, int)

    indica a *borbulha* a espera de um apontador para uma função que recebe 2 *int* e retorna um *bool*.

Apontadores para Funções

- **Exemplo: Ordenação borbulhante (*Bubble Sort*)**
  - **Se os parênteses forem retirados:**

    bool *compare(int, int)

    declarará uma função que recebe dois inteiros e retorna um apontador para um booleano.

Apontadores para Funções

```
01  /* Programa geral para ordenação de dados - uso de apontadores para funções */
02  #include <stdio.h>
03  #define TAMANHO 10
04  void borbulha(int [], const int, int (*)(int, int));
05  int ascendente( int, int );
06  int descendente( int, int );
07  int main()
08  {
09      int ordem, contador, a[TAMANHO] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
10      printf( "Digite 1 para visualizar os dados em ordem crescente,\n "
11              "Digite 2 para visualizar os dados em ordem decrescente " );
12      scanf( "%d ",  &ordem);
13      printf( "\nDados na ordem original\n " );
14      for (contador = 0; contador < TAMANHO ; contador ++)
15         printf( "%5d ", a[contador]);
16      if (ordem == 1) {
17         borbulha(a, TAMANHO , ascendente);
18         printf( "\nDados em ordem crescente\n " );
19      }
20      else {
21         borbulha(a, TAMANHO , descendente);
22         printf( "\nDados em ordem decrescente\n " );
23      }
24     for ( contador = 0; contador < TAMANHO; contador++ )
25         printf( "%5d", a[ contador ] );
26    printf( "\n" );
27    return 0;
28  }
```

Parâmetro apontador para função

Apontadores para Funções

```
29  /* Funções borbulha, permuta,, ascendente e descendente */
30  void borbulha(int b[], const int tam, int (*compara)(int, int))
31  {
32     int passeio, contagem;
33     void permuta(int *, int *);
34     for (passeio = 1; passeio < TAMANHO; passeio ++)
35       for (contagem = 0; contagem < TAMANHO - 1; contage
36         if ((*compara)(b[contagem], b[contagem + 1]))
37           permuta(&b[contagem], &b[contagem + 1]);
38  }
39  void permuta(int *apontelem1, int *apontelem2)
40  {
41     int auxi;
42     auxi  = *apontelem1;
43     *apontelem1 = *apontelem2;
44     *apontelem2 = temp;
45  }
46  int ascendente(int a, int b)
47  {
48     return b < a; }  /*permuta se b for menor do que a*/
49  int descendente(int a, int b)
50  {
51     return b > a;   /*permuta se b for maior do que a*/
52  }
```

*Observar a chamada dos apontadores para funções (uso do **operador de desreferência**. O uso de * não é requerido, porém **compara** é um apontador de função, **não** uma função.*

***ascendente** e **descendente** retornam **verdadeiro** ou **falso**. **borbulha** chama **permuta** se a chamada à função retornar **verdadeiro**.*

Apontadores para Funções

```
Digite 1 para visualizar os dados em ordem crescente
Digite 2 para visualizar os dados em ordem decrescente : 1

Dados na ordem original
  2  6  4  8 10 12 89 68 45 37
Dados em ordem crescente
  2  4  6  8 10 12 37 45 68 89
```

```
Digite 1 para visualizar os dados em ordem crescente
Digite 2 para visualizar os dados em ordem decrescente : 2

Dados na ordem original
  2  6  4  8 10 12 89 68 45 37
Dados em ordem decrescente
 89 68 45 37 12 10  8  6  4  2
```


Porque inicializar apontadores?

- **Observe o código:**

```
main () /* Errado - Nao Execute */
{
  int x,*p;
  x=13;
  *p=x; //posição de memória de p é indefinida!
}
```

- **A não inicialização de apontadores pode fazer com que ele esteja alocando um espaço de memória utilizado, por exemplo, pelo S.O.**

Porque inicializar apontadores?

- No caso de vetores, é necessário sempre alocar a memória necessária para compor as posições do vetor.
- O exemplo abaixo apresenta um programa que compila, porém poderá ocasionar sérios problemas na execução. Como por exemplo utilizar um espaço de memória alocado para outra aplicação.

```
main() {
  char *pc; char str[] = "Uma string";
  strcpy(pc, str);// pc indefinido
}
```

Alocação Dinâmica

- **Objetivos**
  - Utilizar espaços da memória de tamanho arbitrário.
  - Criar estruturas de dados usando encadeamento.
- **Motivação**
  - Alocação de espaço sob demanda: muitas vezes o espaço de memória necessário para um conjunto de dados varia durante a execução do programa.
  - Encadeamento prove um estilo eficiente de representar conjuntos de dados em C e para implementar as estruturas de armazenamento de Tipos Abstratos de Dados

Alocação Dinâmica

- **Alocação Dinâmica**
  - **Declaração de uma variável ⇒ Alocação de memória pelo compilador para armazenamento da variável.**
  - **Alterações de requisitos do programa/ da variável ⇒ Edição para realização das alterações e recompilação do programa.**

Alocação Dinâmica

- **Alocação Dinâmica**
  - **Alocação de espaço de armazenamento *durante* a execução ⇨ Redução no número de alterações nos tamanhos das variáveis.**
  - **Alocação Dinâmica ⇒ Retorno de um apontador para o início do intervalo de memória pela biblioteca de execução de C.**

Alocação Dinâmica

- **Alocação Dinâmica**
  - **Alocação de memória *durante* a execução ⇨ Uso da função *malloc*.**
  - **Sintaxe:**

    #include <alloc.h>

    void *malloc(size_t num_bytes);

    ***num_bytes* ⇨ número de bytes a ser alocado**

Alocação Dinâmica

- **Uso de *malloc***
  - **Satisfatório ⇨ Retorno de um apontador para o início do intervalo.**
  - **Insatisfatório ⇨ Retorno de *NULL* (ocorrência de erro).**

Alocação Dinâmica

- **Exemplo 1 – Uso de *malloc***

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

void main(void)
{
  char *string;
  int *int_dados;
  float *float_dados;

  if ((string = (char *) malloc(50)))
    printf( "Sucesso na alocação de uma string de 50 bytes\n");
  else
    printf("Erro na alocação da string\n");

  if ((int_dados = (int *) malloc(100 * sizeof(int))) != NULL)
    printf(" Sucesso na alocação de int_dados[100]\n");
  else
    printf(" Erro na alocação de int_dados[100]\n");

  if ((float_dados = (float *) malloc(25 * sizeof(float))) != NULL)
    printf(" Sucesso na alocação de float_dados[25]\n");
  else
    printf(" Erro na alocação de float_dados[25]\n");
}
```


Alocação Dinâmica

- **Uso de *malloc* – Conversão do tipo de apontador**
  - **Alocação de um apontador para *n* dados do tipo *int:***

    int *dados;

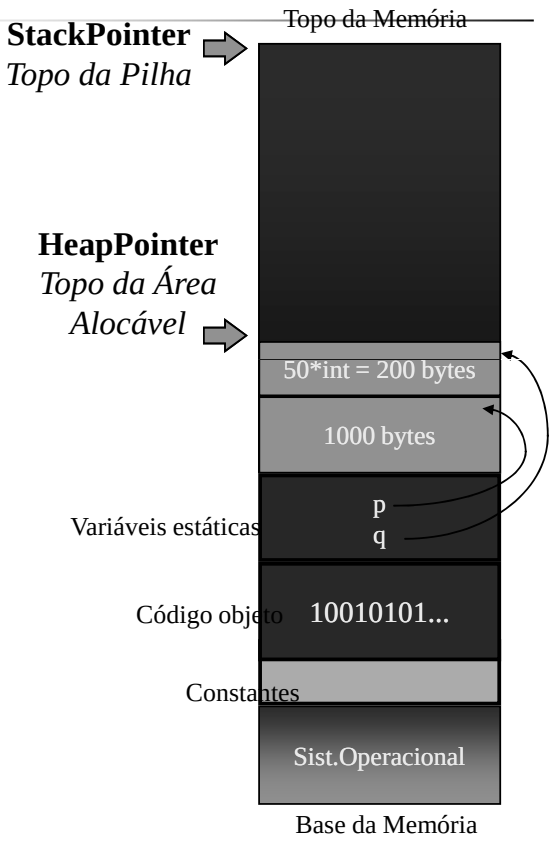
    dados = (int *) malloc(n * sizeof(int))
  - **Alocação de um apontador para *k* dados do tipo *float:***

    float *dados;

    dados = (float *) malloc(k * sizeof(float))

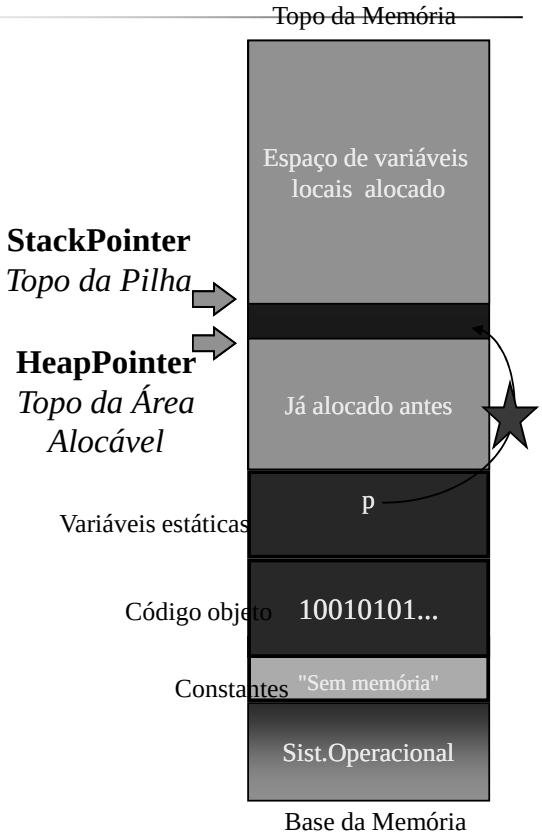
Exemplo:

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

char    *p;
int     *q;

main ()
{
   p = (char *) malloc(1000);
          // Aloca 1000
          // bytes de RAM

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



- malloc devolve:
  - um ponteiro para a área alocada
  - o ponteiro nulo (NULL) caso não seja possível alocar a memória requisitada.
- Convém verificar se foi pos-sível alocar a memória:

```
#include <stdio.h>
#include <stdlib.h>
char    *p;
main ()
{  ................
   p = malloc(1000)
          // Tenta alocar 1000
          // bytes de RAM
   if (p == NULL)
          // Testa se p
          // diferente de 0
         printf("Sem memória.");
}
```



Alocação Dinâmica

- **Liberação da memória alocada após o uso**
  - **Chamada da função *free()*:**

    #include <alloc.h>

    void free(apont);

    apont ⇨ **apontador para o início do intervalo de memória a ser liberado**

Alocação Dinâmica

- **Função *free()*:**
  - **Devolve memória previamente alocada ao sistema.**
  - **A memória devolvida é aquela que foi alocada com um ponteiro com o valor de `apont`.**
    - **O valor de `apont` deve ser um valor que foi alguma vez retornado por malloc().**
    - **Não é possível alocar-se um vetor enorme e depois dealocar-se a parte dele que "sobrou".**
  - **A utilização de `free`() com um valor de ponteiro qualquer poder ter resultados catastróficos.**
  - **A gerência de *buracos* no heap é responsabilidade do sistema operacional.**

Alocação Dinâmica

- Uso da função ***calloc()***
  - ***malloc()*** ⇨ Número de bytes a ser alocado.
  - ***calloc()*** ⇨ Número de elementos de um tamanho específico a serem alocados.
    - Sintaxe:

```
#include <alloc.h>
void *calloc(size_t num_itens, size_t tam_item);
```

num_itens ⇨ Número de itens a serem alocados
tam_item ⇨ Tamanho (em bytes) de cada item

Alocação Dinâmica

- Exemplo 2 – Uso de ***calloc***

```
#include <stdio.h>
#include <alloc.h>
void main(void)
{
  char *string;
  int *int_dados;
  float *float_dados;
  if ((string = (char *) calloc(50, sizeof(char))))
    printf( "Sucesso na alocação de uma string de 50 bytes\n");
  else
    printf("Erro de alocação da string\n");
  if ((int_dados = (int *) calloc(100, sizeof(int))) != NULL)
    printf(" Sucesso na alocação de int_dados[100]\n");
  else
    printf(" Erro de alocação de int_dados[100]\n");
  if ((float_dados = (float *) calloc(25, sizeof(float))) != NULL)
    printf(" Sucesso na alocação de float_dados[25]\n");
  else
    printf(" Erro de alocação de float_dados[25]\n");
}
```

Alocação Dinâmica

- Uso de ***calloc***
  - Satisfatório ⇨ Retorno de um apontador para o início do intervalo.
  - Insatisfatório ⇨ Retorno de ***NULL*** (ocorrência de erro).
  - Liberação após o uso a partir da função ***free()***.

Se o programa NÃO usar *free()* para liberar a memória após o uso, a liberação ocorrerá automaticamente. Todavia, é ACONSELHÁVEL liberar a memória assim que esta não mais for necessária!

Alocação Dinâmica

- Diferença entre a função ***malloc()*** e a função ***calloc()***
  - O *calloc* zera todos os bits da memória alocada enquanto que o *malloc* não.
  - Logo, se não for necessário uma inicialização (com zero) da memória alocada, o *malloc* é preferível por ser um pouco mais rápido.

Alocação Dinâmica

Alocação dinâmica de vetores

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

float *Alocar_vetor_real (int n)
{
  float *v;     /* ponteiro para o vetor */
  if (n < 1)
     printf ("** Erro: Parametro invalido
**\n");
     return (NULL);
     }
  /* aloca o vetor */
  v = (float *) calloc (n+1, sizeof(float));
  if (v == NULL) {
     printf ("* Erro: Memoria Insuficiente
**");
     return (NULL);
  }
  else {
     printf ("** \n\nMemoria Alocada com
Sucesso\n\n **");
  }
  return (v);
  /* retorna o ponteiro para o vetor */
}
```

```
float *Liberar_vetor_real (int n, float *v)
{
  if (v == NULL) return (NULL);
  if (n < 1) { /* verifica parametros
recebidos */
     printf ("** Erro: Parametro invalido
**\n");
     return (NULL);
     }
  free(v);        /* libera o vetor */
  return (NULL);  /* retorna o ponteiro */
}

void main (void)
{
  float *p;
  int a;
  printf("\nDigite o tamanho do vetor-->");
  scanf("%d", &a);
  ...    /* outros comandos */
  p = Alocar_vetor_real (a);
  ...     /* outros comandos, utilizando p[]
normalmente */
  p = Liberar_vetor_real (a, p);
}
```


Alocação Dinâmica

Alocação dinâmica de matrizes

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

float **Alocar_matriz_real (int m, int n)
{
  float **v;  /* ponteiro para a matriz */
  int   i;    /* variavel auxiliar      */
  if (m < 1 || n < 1) { /* verifica
parametros recebidos */
     printf ("** Erro: Parametro invalido
**\n");
     return (NULL);
  }
  /* aloca as linhas da matriz */
  v = (float **) calloc (m, sizeof(float
*));
  if (v == NULL) {
     printf ("** Erro: Memoria Insuficiente
**");
     return (NULL);
  }
  /* aloca as colunas da matriz */
  for ( i = 0; i < m; i++ ) {
   v[i]     =     (float*)     calloc     (n,
sizeof(float));
   if (v[i] == NULL) {
       printf     ("**     Erro:     Memoria
Insuficiente **");
         return (NULL);
   }
  }
 return (v); /*retorna pont para matriz */
```

```
float **Liberar_matriz_real (int m, int n,
float **v)
{
  int  i;  /* variavel auxiliar */
  if (v == NULL) return (NULL);
  /* verifica parametros recebidos */
  if (m < 1 || n < 1) {
     printf ("** Erro: Parametro invalido
**\n");
     return (v);
  }
  /* libera as linhas da matriz */
  for (i=0; i<m; i++) free (v[i]);
  free (v);      /* libera a matriz */
  /* retorna um ponteiro nulo */
  return (NULL);
}

void main (void)
{
  float **mat;  /* matriz a ser alocada */
  int     l, c;     /* numero de linhas e
colunas da matriz */
  ...                  /* outros comandos,
inclusive inicializacao para l e c */
  mat = Alocar_matriz_real (l, c);
  ...                  /* outros comandos
utilizando mat[][] normalmente */
  mat = Liberar_matriz_real (l, c, mat);
  ...
}
```

Alocação Dinâmica

- **Alocação dinâmica de matrizes**
  - *Exemplo com malloc*

```
#float** alocarMatriz(int Linhas,int Colunas){
//Recebe a quantidade de Linhas e Colunas como Parâmetro

  int i,j; //Variáveis Auxiliares

  //Aloca um Vetor de Ponteiros
  float**m = (float**)malloc(Linhas * sizeof(float*));

  //Percorre as linhas do Vetor de Ponteiros
  for (i = 0; i < Linhas; i++){

    //Aloca um Vetor de Floats para cada posição do Vetor de Ponteiros.
    m[i] = (float*) malloc(Colunas * sizeof(float));

       //Percorre o Vetor de Floats atual.
       for (j = 0; j < Colunas; j++){

            //Inicializa com 0.
            m[i][j] = 0; }
  }
  //Retorna o Ponteiro para a Matriz Alocada
  return m;
}
```


Alocação Dinâmica

- **Uso da função *realloc()***
  - ***realloc()*** ⇨ **faz um bloco já alocado crescer ou diminuir, preservando o conteúdo já existente.**
    - **Sintaxe:**

#include <alloc.h>

void *realloc(*apont, size_t novo_tam);

apont ⇨ **apontador para o início do intervalo de memória a ser liberado**

novo_tam ⇨ **Novo tamanho (em bytes) de cada item**

Alocação Dinâmica

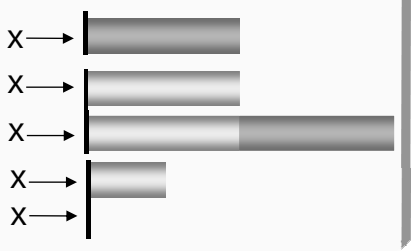
Uso da função *realloc()*

- **A funçao *realloc()* modifica o tamanho da memória previamente alocada apontada por *apont* para aquele novo valor especificado por *novo_tam*. O valor de *novo_tam* pode ser maior ou menor que o original.**
- **Um ponteiro para o bloco é devolvido porque *realloc()* pode precisar mover o bloco para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado no novo bloco, e nenhuma informação é perdida.**
- **Se *apont* for nulo, aloca size bytes e devolve um ponteiro; se size é zero, a memória apontada por *apont* é liberada.**
- **Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.**

Heap

Exemplo 3 – Alocação com *realloc()*

```
#include <stdio.h>
#include <alloc.h>
void main(void)
{
  int *x, i;
  x = (int *) malloc(4000*sizeof(int));
  for(i=0;i<4000;i++)
    x[i] = rand()%100;
  x = (int *) realloc(x, 8000*sizeof(int));
  x = (int *) realloc(x, 2000*sizeof(int));
  free(x);
  getch();
}
```

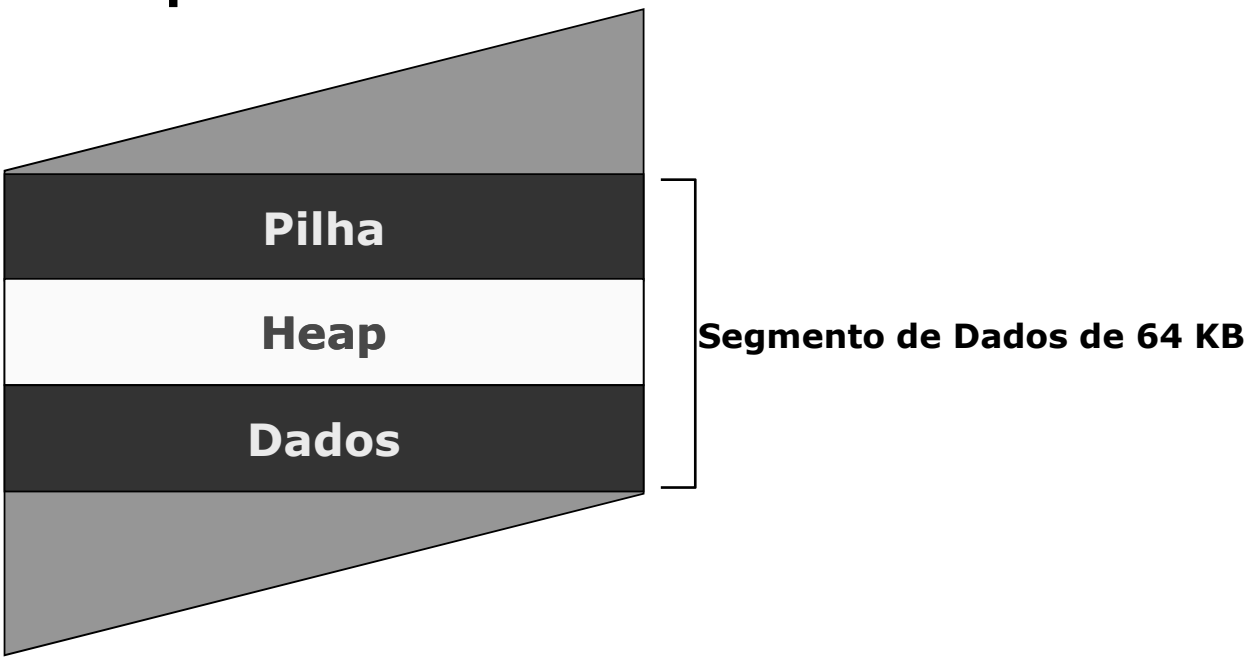


Heap

- **Heap**
  - **Reservatório de memória não usado, a partir do qual a biblioteca de execução de C reserva memória quando programas alocam dinamicamente a memória.**
  - **Modelo de memória *small* ⇨ Área de memória entre o topo da área de dados do programa compilado e a pilha.**

Heap

- **Heap**



Heap

- **Heap**
  - **Residente no segmento de dados do programa.**
  - **Quantidade de heap disponível para um programa ⇨ Fixa.**
  - **Falha de alocação para quantidades de memória superiores a 64 KB.**

Heap

- **Exemplo 4 – Falha de alocação por falta de espaço**

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

void main(void)
{
  char *m1, *m2, *m3;

  if ((m1 = (char *) malloc(30000)) == NULL)
    printf("Erro de alocação da matriz 1\n");
  else if ((m2 = (char *) malloc(30000)) == NULL)
    printf("Erro de alocação da matriz 2\n");
  else if ((m3 = (char *) malloc(30000)) == NULL)
    printf("Erro de alocação da matriz 3\n");
  else
    printf("Sucesso na alocação de todas as matrizes \n");
}
```

Heap

- Contorno da Limitação de 64 KB para o Heap.
  - Uso de ***farmalloc()*** e ***farcalloc()***:
    - Parâmetros idênticos àqueles passados para ***malloc()*** e ***calloc()***.
    - Necessidade de uso de um ponteiro *far* para os dados.

Heap

- Exemplo 5 – Alocação com ***farmalloc()***

```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>

void main(void)
{
  char *string;
  int *int_dados;
  float *float_dados;

  clrscr();
  if ((string = (char far*) farmalloc(50000)))
    printf("Sucesso na alocação de uma string de 50 Kbytes\n");
  else printf("Erro na alocação da string\n");

  if((int_dados=(int far*) farmalloc(100000*sizeof(int)))!= NULL)
    printf(" Sucesso na alocação de int_dados[100000]\n");
  else printf(" Erro na alocação de int_dados[100000]\n");

  if((float_dados=(far float*)farmalloc(250*sizeof(float)))!= NULL)
    printf(" Sucesso na alocação de float_dados[250]\n");
  else printf(" Erro na alocação de float_dados[250]\n");
  getch();
}
```

Apontadores para tipos estruturados

- Apontadores são normalmente utilizados com tipos estruturados

```
Typedef struct {
        int idade;
        double salario;
} TRegistro;

TRegistro *a;
...
a = (TRegistro *)
malloc(sizeof(TRegistro))
a->idade = 30;   /* *a.idade = 30 */
a->salario = 80;
```

Apontadores para tipos estruturados

```
struct data{
   int dia;
   int mês;
   int ano;
   };
```

- Definindo uma variável do tipo data:
  `struct data dt;`
- Definindo um ponteiro para dt:
  `struct data *pdt = &dt;`
- Fazendo referência a um elemento da estrutura:
  `dt.dia` ou `(*pdt).dia` ou `pdt->dia`
  `dt.mes` ou `(*pdt).mes` ou `pdt->mês`
  `dt.ano` ou `(*pdt).ano` ou `pdt->ano`

Apontadores para tipos estruturados

Exemplo de programa demonstrando uso de ponteiro, passagem de parâmetro para função e estrutura

```
#include <stdio.h>
typedef struct estrutura{
       int valor;
} ESTRUTURA;

void incrementa(ESTRUTURA * e){
     printf("    Valor --> %i \n", e->valor);
     printf("    Valor --> %i \n", (*e).valor);  /* outra
forma de acessar */
       (e->valor)++;       /* incrementa valor */
}

int main(){
      ESTRUTURA d;         /* declara variavel struct */
      d.valor = 0;         /* atribui valor inicial */
      printf("Antes da funcao --> %d \n", d.valor);
      incrementa(&d);      /* invoca a funcao */
      printf("Apos a funcao --> %d\n", d.valor);
        return 0;
}
```

Apontadores para tipos estruturados

- **É preciso armazenar esse endereço retornado pela malloc num ponteiro de tipo apropriado**
- **Para alocar um tipo de dado que ocupa vários bytes, é preciso recorrer ao operador sizeof, que diz quantos bytes o tipo especificado tem**
- **Vamos utilizar outra forma de declaração da struct:**

```
struct tipo_data{
        int dia, mes, ano;
     };
typedef struct tipo_data
DATA;
```

ATENÇÃO: data é o nome do

Outra forma de declaração:

```
typedef struct tipo_data{
     int dia, mes, ano;
} DATA;
```

```
DATA *d;
d = malloc (sizeof (DATA));
d->dia = 31;
d->mes = 12;
d->ano = 2008;
```

Erros Comuns na Alocação Dinâmica

- **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.**

Exercícios

1. **Faça um programa que leia um valor *n*, crie dinamicamente um vetor de *n* elementos e passe esse vetor para uma função que vai ler os elementos desse vetor.**

2. **Declare um TipoRegistro, com campos *a* inteiro e *b* que é um apontador para char. No seu programa crie dinamicamente uma variável do TipoRegistro e atribua os valores 10 e 'x' aos seus campos.**

Solução (1)

```
void LeVetor(int *a, int n){
    int i;
    for(i=0; i<n; i++)
        scanf("%d",&a[i]);
}
int main(int argc, char *argv[]) {
   int *v, n, i;
   scanf("%d",&n);
   v = (int *) malloc(n*sizeof(int));
   LeVetor(v,n);
   for(i=0; i<n; i++)
       printf("%d\n",v[i]);
   free(v);
}
```

Apesar do conteúdo ser modificado Não é necessário passar por referência pois todo vetor já é um apontador...

Solução (2)

```
typedef struct {
   int a;
   char *b;
} TRegistro;

int main(int argc, char *argv[])
{
   TRegistro *reg;
   reg = (TRegistro *) malloc(sizeof(TRegistro));
   reg->a = 10;
   reg->b = (char *) malloc(sizeof(char));
   *(reg->b) = 'x';
   printf("%d %c",reg->a, *(reg->b));
   free(reg->b);
   free(reg);
}
```

É necessário alocar espaço para o registro e para o campo b. *(reg->b) representa o conteúdo da variável apontada por reg->

Faça:

- **Criar um tipo que é uma estrutura que represente uma pessoa, contendo nome, data de nascimento e CPF.**
- **Criar uma variável que é um ponteiro para esta estrutura (no programa principal)**
- **Criar uma função que recebe este ponteiro e preenche os dados da estrutura**
- **Criar uma função que recebe este ponteiro e imprime os dados da estrutura**
- **Fazer a chamada a estas funções na função principal**