SIN 211 - Algoritmos e Estruturas de Dados
(Revisão 2)

Profº: Joelson Antônio dos Santos

Universidade Federal de Viçosa
Instituto de Ciências Exatas e Tecnológicas
Campus de Rio Paranaíba - MG

joelsonn.santos@gmail.com
Sala: BBT 233

15 de março de 2018

Roteiro

1. Variáveis e Ponteiros
2. Ponteiro para ponteiro
3. Passagem de parâmetro por valor e por referência
4. Alocação de memória
   - Alocação estática
   - Alocação dinâmica

Créditos

O material desta aula é composto por adaptações e extensões dos originais gentilmente cedidos pelos professores **Moacir Pereira Ponti** e **Rachel Reis**.

Aula anterior

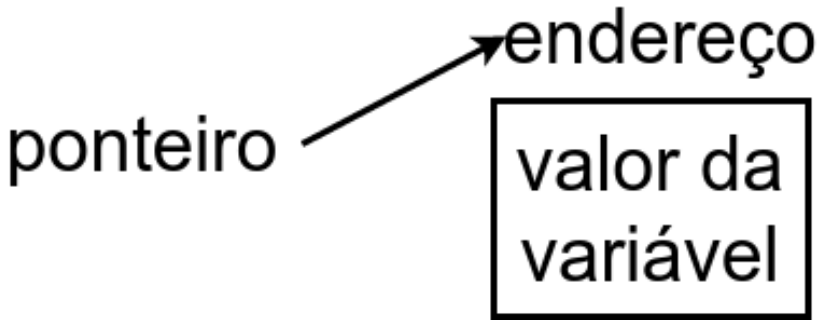
- *Arrays* e Estruturas
  - Definição
  - Declaração
  - Impressão

Relembrando - Variável

- Região da memória representada por um nome e dedicada a armazenar informação.
- Possui um tipo de dados (à ser armazenado) determinado no momento de sua criação.
- O conteúdo de uma variável pode ser manipulado.

Ponteiro

- É uma variável que armazena o **endereço de memória de outra variável**.
- Este endereço normalmente é de uma outra variável que está em uma outra posição qualquer da memória.



Ponteiro

- Se uma variável ponteiro armazena o endereço de memória de outra variável, então é comum de dizer que a variável ponteiro aponta para a outra variável.
- Ponteiro possui tipo, pois também é uma variável.
- O tipo de um ponteiro é sempre ponteiro ($*$) para o tipo de dado da região de memória (variável) onde esse ponteiro aponta.

Ponteiro - Declaração

- Forma geral de declaração de ponteiros:

  tipo_de_dado* nome_ponteiro;

- Alguns exemplos:

```
int* ponteiro1;
double* ponteiro2;
struct Estrutura* ponteiro3;
```

Ponteiro

- O valor inicial de um ponteiro é sempre lixo de memória, por isso é recomendado sempre inicializá-lo com **NULL**.

- Cuidado ao tentar utilizar um ponteiro cujo valor é **NULL**, pois não é possível armazenar um valor onde este ponteiro aponta.

Ponteiro - Cuidados

OBS: O que está destacado de vermelho é uma instrução incorreta. Em contrapartida, o que está destacado como verde é determinado como uma instrução correta nos exemplos.

```
int a = 5;
int* p = NULL;
*p = a;
```

```
int a = 5;
int* p = NULL;
*p = &a;
```

```
int a = 5;
int* p = NULL;
p = &a;
```

```
int* p;
p = (int*)
malloc(sizeof(int));
*p = 5;
```

Operadores de ponteiros

- Existem 2 operadores utilizados com ponteiros: $*$ e &

- O operador & retorna o endereço de memória de uma variável que um determinado ponteiro aponta.

  Exemplo:

  ptr = &var1;

- O operador $*$ retorna o valor da variável apontada pelo ponteiro.

Operadores de ponteiros

- Note que o operador & é geralmente usado antes de uma variável, pois ele retorna o endereço de memória dela.

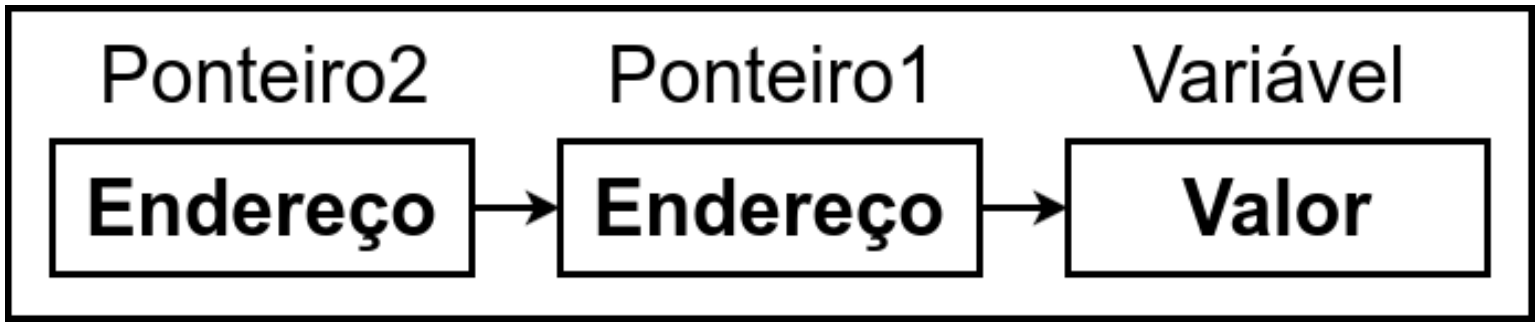
  ptr = &var1;

- Note que o operador $*$ é geralmente usado antes de um ponteiro, pois ele retorna o valor da variável apontada pelo ponteiro.

  var2 = *ptr;

Ponteiro para Ponteiro

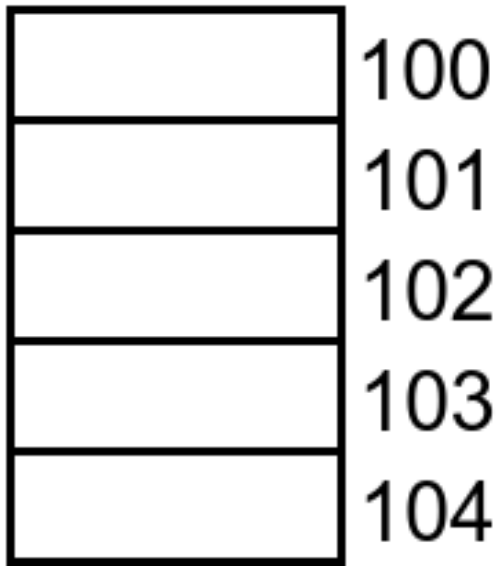
- Existem situações em que um ponteiro pode estar apontando para **outro ponteiro** que aponta para o **valor final** (endereço de uma variável comum).
- Essas situações são chamadas de **indireção múltipla** ou **ponteiro para ponteiro**.



Ponteiro para Ponteiro - Exemplo

Atribuições

Memória

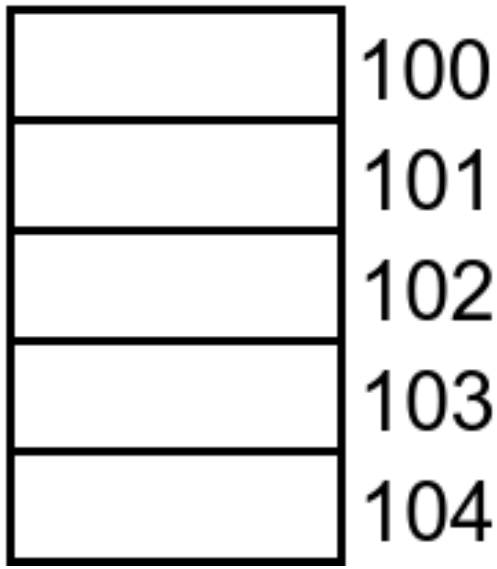


Ponteiro para Ponteiro - Exemplo

Atribuições

```
var1 = 10;
```

Memória

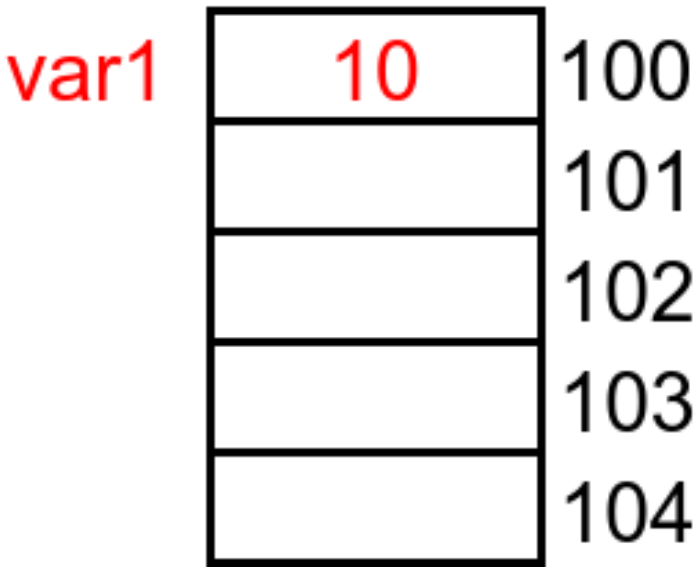


Ponteiro para Ponteiro - Exemplo

Atribuições

```
var1 = 10;
```

Memória

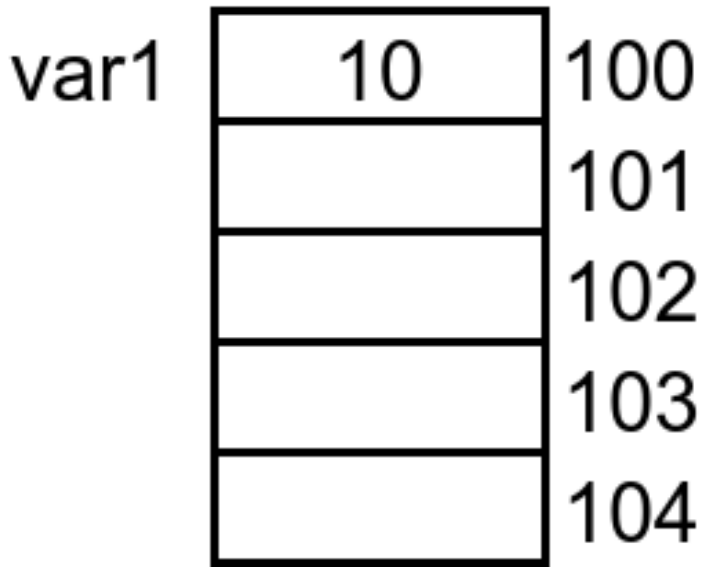


Ponteiro para Ponteiro - Exemplo

Atribuições

```
var1 = 10;
ptr1 = &var1;
```

Memória



Ponteiro para Ponteiro - Exemplo

Atribuições

```
var1 = 10;
ptr1 = &var1;
```

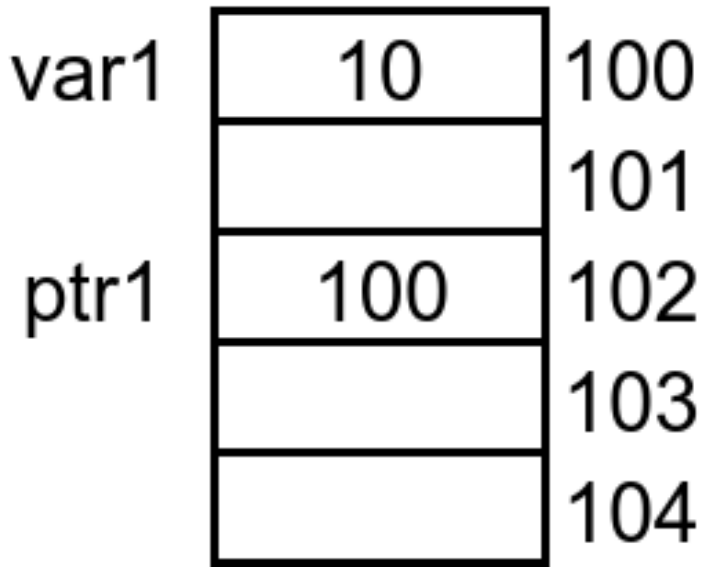
Memória

Ponteiro para Ponteiro - Exemplo

Atribuições

```
var1 = 10;
ptr1 = &var1;
ptr2 = &prt1;
```

Memória

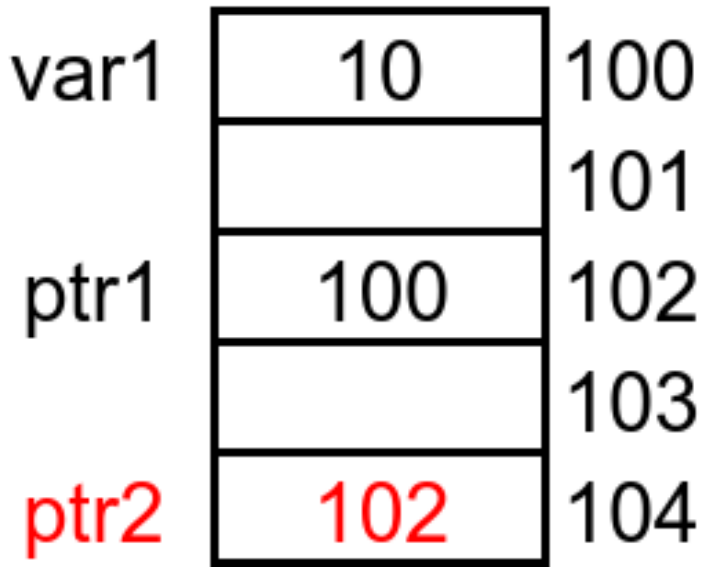


Ponteiro para Ponteiro - Exemplo

Atribuições

```
var1 = 10;
ptr1 = &var1;
ptr2 = &prt1;
```

Memória



Ponteiro para Ponteiro - Declaração

- Forma geral de declaração de ponteiro para ponteiro:

  tipo_de_dado** nome_ponteiro_para_ponteiro;

- Alguns exemplos:

  int** ponteiro_para_ponteiro1;
  double** ponteiro_para_ponteiro2;
  struct Estrutura** ponteiro_para_ponteiro3;

- Para declarar um ponteiro para ponteiro, utiliza-se **.

Ponteiro para Ponteiro - Atribuir valores

- Um ponteiro para ponteiro também é um ponteiro e portanto se comporta da mesma maneira, exceto pelo conteúdo apontado.

- Ponteiro para ponteiro só aceita/recebe a atribuição de um endereço de memória de outro ponteiro, e não de uma variável comum.

Ponteiro para Ponteiro - Sintaxe

- Dentro da função *main*:

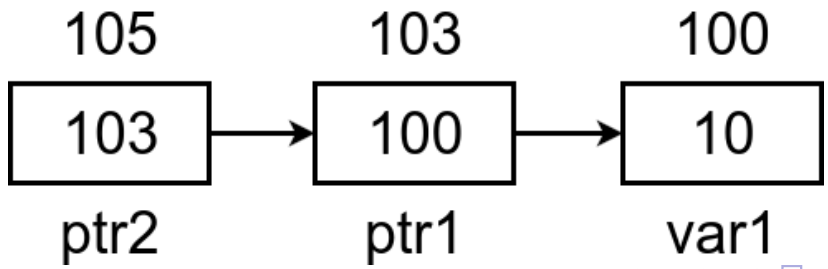
```
int var1 = 10;          variável do tipo int.
int* ptr1;              ponteiro
int** ptr2;             ponteiro para ponteiro
ptr1 = &var1;           ptr1 recebe o endereço de
memória de var1
ptr2 = &ptr1;           ptr2 recebe o endereço de
memória de ptr1

printf("Valor de var1: %d ", **ptr2);
Resultado do printf igual a 10.
```

Ponteiro para Ponteiro - Impressão

- Dentro da função *main*:

printf("var1: %d ", var1);	a: 10.
printf("&a: %p ", &var1);	&a: 100.
printf("ptr1: %p ", ptr1);	ptr1: 100.
printf("&ptr1: %p ", &ptr1);	&ptr1: 103.
printf("ptr2: %p ", ptr2);	ptr2: 103.
printf("&ptr2: %p ", &ptr2);	&ptr2: 105.



Ponteiros e Estruturas

- As variáveis do tipo estruturas são tratadas da mesma maneira que variáveis de tipos primitivos (e.g., int, double, ...).

- É possível definir variáveis do tipo ponteiro para estruturas.

Ponteiros e Estruturas

- As variáveis do tipo estruturas são tratadas da mesma maneira que variáveis de tipos primitivos (e.g., int, double, ...).
- É possível definir variáveis do tipo ponteiro para estruturas.

Exemplo: Endereco* ptrEnd;

```
typedef struct sEndereco{
    char rua[40];
    int numero;
    char bairro[40];
    char cidade[30];
    char estado[3];
    long int CEP;
}Endereco;
```

Ponteiros para Estruturas

```c
#include <stdio.h>
#include <string.h>

typedef struct sEndereco{
    char rua[40];
    int numero;
    char bairro[40];
    char cidade[30];
    char estado[3];
    long int CEP;
}Endereco;


int main ()
{

   Endereco end; // variável comum do tipo Endereco
   Endereco *ptrEnd; // ponteiro do tipo Endereco

   ptrEnd = &end; // ptrEnd recebe o endereço de memória de end
   strcpy(end.rua, "Rua A");
   strcpy(ptrEnd->cidade, "Rio Paranaiba");

   printf("%s", end.rua);
   printf("\n%s", ptrEnd->cidade);

   return(0);
}
```

Ponteiros para Estruturas - Formas de acesso aos campos individuais

- Existem formas de acesso a campos individuais de ponteiros de estruturas, duas delas são:

(*ptrEnd).numero = 200;

ou

ptrEnd− >numero = 100;

Parâmetro de função

- Como uma função pode alterar o valor das variáveis da função que as chamou?

Parâmetro de função

- Em linguagem C, existem duas maneiras de passagem de parâmetros para uma determinada função:
  - **Por valor:** onde é passado uma **cópia do valor original**.
  - **Por referência:** é passado o **endereço de memória do valor original**.

Parâmetro de função

- Na passagem de parâmetros por **valor**, as variáveis passadas a uma função sofrem alterações em seus valores apenas na função que as chamou. Logo, **não altera o valor original passado**.

- Na passagem de parâmetros por **referência:**, o que realmente é passado são os endereços de memórias das variáveis.
  - Utiliza-se ponteiros nos argumentos das funções que recebem endereços de memória de variáveis.

  - Uma vez passado o endereço de qualquer variável para uma determinada função, **o valor da variável original está sujeito a alterações**.

Passagem de parâmetros por referência

```c
#include <stdio.h>
#include <string.h>


void altera(int *pa, int *pb){
    *pa = 4;
    *pb = 32;
}

int main(){
    int x = 0, y = 0;
    altera(&x, &y);
    printf("%d %d ", x, y);
    return 0;
}
```

Alocação Estática de Memória

- As variáveis de um programa têm alocação estática se a quantidade total de memória utilizada pelos dados é previamente conhecida e definida de modo imutável, no próprio código-fonte do programa.

- Na linguagem C, o espaço de memória utilizado por um programa para armazenar dados normalmente é indicado pelo programador no momento da declaração de variáveis.

  Exemplo: `int vec[10];`

Alocação Estática de Memória

- Vantagens:
  - Sequencial; Indexação eficiente; quaisquer posições de matrizes ou vetores são de fácil acesso.
- Desvantagens:
  - É necessário saber previamente o tamanho máximo do (vetor ou matriz) a ser declarado.
  - Pode proporcionar desperdício de memória.
  - Inserção e remoção de elementos são custosas computacionalmente.

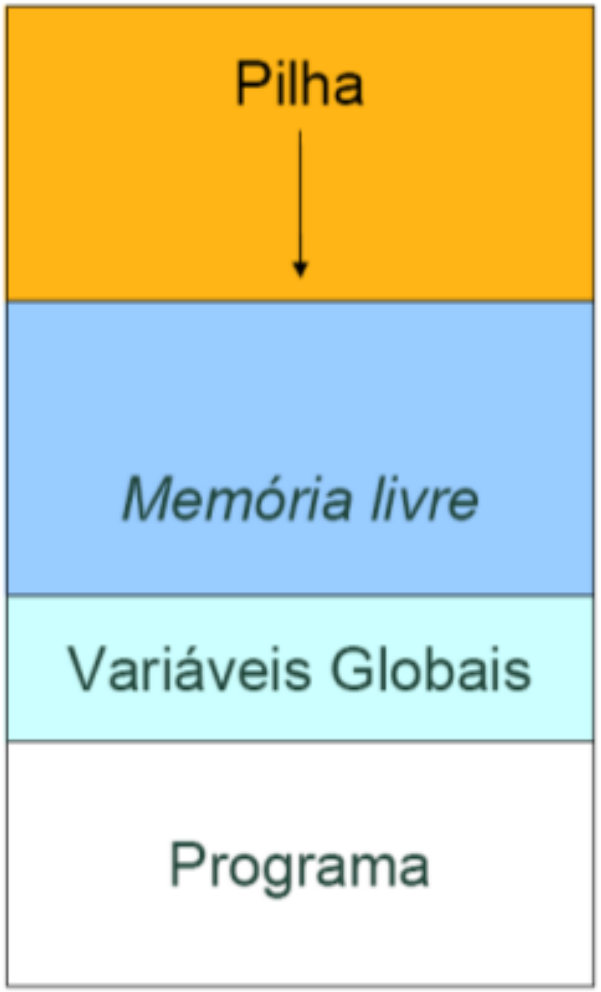
    Exemplo: inserir elemento na posição 0 de um vetor que já contém alguns outros elementos nas primeiras posições. (Deslocamento dos elementos para a inserção)

Alocação Dinâmica de Memória

- As variáveis de um programa têm alocação dinâmica quando suas **áreas de memória**, não declaradas no programa, **passam a existir durante a execução**. Em outras palavras, o programa é capaz de criar novas variáveis enquanto executa.
- Como utilizar alocação dinâmica?
  - Uso de ponteiros.
  - funções de reserva e liberação de espaços de memória em tempo de execução.

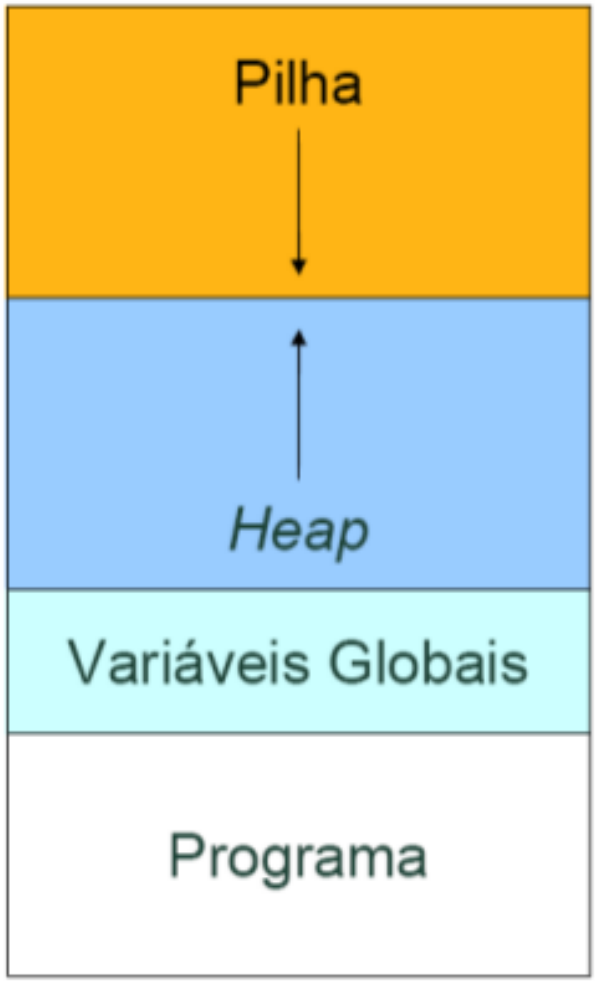
Uso da Memória Principal (RAM)

- A memória principal pode ser usada para armazenar variáveis locais e globais (e.g., *arrays* e estruturas).
  - **Variáveis locais:** o armazenamento é feito na **pilha do computador**.
  - **Variáveis globais:** o **armazenamento é fixo** durante todo o tempo de execução do programa.



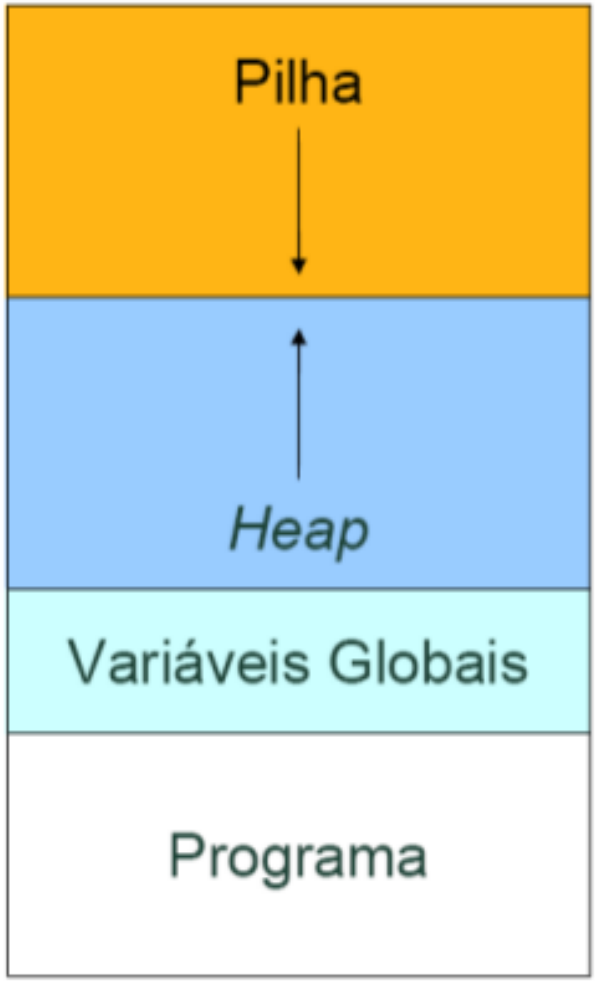
Alocação Dinâmica de Memória

- Para oferecer um meio de armazenar dados em **tempo de execução, há um subsistema de alocação dinâmica.**
- O armazenamento de forma dinâmica é feito na região de memória livre, chamada de **Heap**.



Alocação Dinâmica de Memória

- A pilha cresce em direção inversa ao *Heap*.
  - Conforme o uso de variáveis na pilha e a alocação de recursos no *Heap*, a memória poderá se esgotar e um novo pedido de alocação de memória falhar.



Alocação Dinâmica de Memória

- No núcleo do sistema de alocação dinâmica em C estão as funções:
  - **malloc()**
  - **free()**

- A função **malloc()** é utilizada para alocar uma porção de memória que estiver livre.

- A função **free()** é usada para devolver ao sistema uma porção de memória alocada que não será mais utilizada pelo programa[1].

[1] Essas funções podem ser encontradas na biblioteca **stdlib.h**

Função *sizeof*

- A função **sizeof(tipo)** recebe um tipo de dado por parâmetro e retorna um número inteiro equivalente à quantidade em **bytes** que aquele tipo de dado ocupa na memória.

  sizeof(int);

- É utilizada em conjunto com a função *malloc*

Função *malloc*

- sintaxe: ptr = **malloc**(numero_de_bytes_alocados);
- A função **malloc** retorna o endereço de memória do primeiro *byte* de uma região de memória do tamanho (numero_de_bytes_alocados) alocada.
- Se não há memória disponível, **malloc** retorna **NULL**.
- **Exemplo:**

```
int* pi
// aloca espaço para um inteiro.
pi = (int*) malloc(sizeof(int));
```

Usando a função *malloc*

- Uma porção de memória capaz de guardar *n* números reais (float) é reservada, ficando o ponteiro **pt** a apontar para o endereço inicial dessa porção de memória.
- O cast da função **malloc()** - (float) - garante que endereço de memória retornado é para o tipo especificado na declaração do ponteiro.
  - **Alguns compiladores requerem o cast obrigatoriamente.**

```
// Trecho de código

float * pt;   // ponteiro
int n;        // variável

printf("Qtd. Valores?");
scanf("%d", &n);

pt = (float*)
   malloc(n*sizeof(float));

if(pt != NULL) {
   // Manipula região
   // alocada
}
```

Função *free*

- A função **free()** é utilizada para liberar o armazenamento de uma variável alocada dinamicamente.
- A função **free()** devolve ao *Heap* a memória apontada pelo ponteiro **pt**, liberando memória.
- Deve ser chamada apenas com ponteiro previamente alocado.
- Toda memória não mais utilizada deve ser desalocada.

```
int* pt;
pt = (int*)
malloc(sizeof(int));
free(pt);
```

Alocação Dinâmica - Exemplo 1

```
int main ()
{
    int *p;
    p = (int *) malloc( sizeof(int) );
    if ( p == NULL )
    {
        printf("Não foi possível alocar memória.\n");
        exit(1);
    }
   *p = 5;
    printf("%d\n", *p);
    free(p);
    return 0;
}
```

Alocação Dinâmica - Exemplo 2

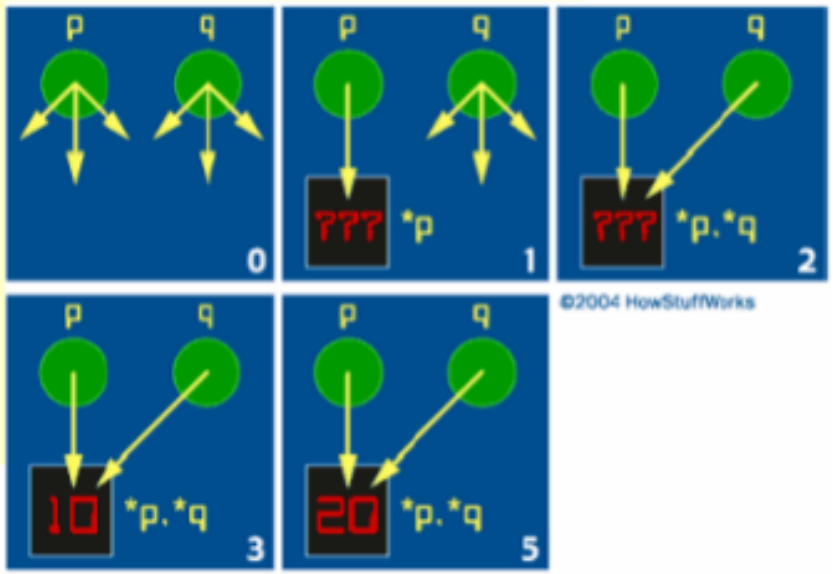
(tenta alocar uma estrutura)

```
struct sEndereco
{
  char rua[20];
  int numero;
};
```

```
int main ()
{
  struct sEndereco *pend;
  pend = (struct sEndereco *)malloc( sizeof(struct sEndereco) );
  if ( pend == NULL )
  {
       printf("Não foi possível alocar memória.\n");
       exit(1);
  }
  return 0;
}
```

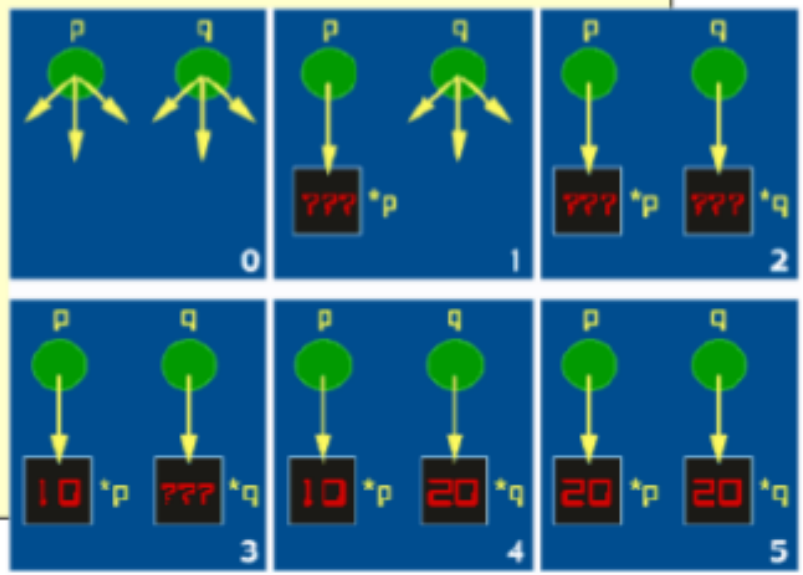
Alocação Dinâmica - Exemplo 3

```
int main (void) {
   int *p, *q;
   p = (int *) malloc(sizeof(int));
   q = p;
   *p = 10;
  *q = 20;
   free(p);
   q = NULL;
   return 0;
}
```



Alocação Dinâmica - Exemplo 4

```
int main (void) {
  int *p, *q;
  p = (int *) malloc(sizeof(int));
  q = (int *) malloc(sizeof(int));
  *p = 10;
 *q = 20;
  *p = *q;
  free(p);
  free(q);
  return 0;
}
```



Alocação Dinâmica - Exemplo 5

```c
int main ()
{
   int *array1;
   array1 = malloc(100 * sizeof(int));
   if (array1 == NULL )
   {
      printf("Não foi possível alocar memória.\n");
      exit(1);
   }
   array1[99] = 301;
   printf("%d\n", array1[99]);
   free(array1);
   return 0;
}
```

Alocação Dinâmica - Exemplo 6

```
int main () {
  CLIENTE *pc;
  pc = (CLIENTE *) malloc( 50 * sizeof(CLIENTE) );

  gets( pc[0].nome );
  scanf("%d", &pc[0].idade );

  printf("%s", pc[0].nome);
  printf("%d", pc[0].idade);

  free(pc);
  return 0;
}
```

```
typedef struct cli
{
   char nome[30];
   int idade;
} CLIENTE;
```

Bibliografia Básica

- SCHILDT H. C Completo e Total. Ed. Makron Books, 3ª edição, 1997
  - Ponteiros: pag. 113
  - Alocação dinâmica: pag: 128
  - funções: pag. 138
  - estrutura: pag. 167

- FORBELLONE, A.L.V.; EBERSPACHER, H.F. Lógica de Programação: a construção de algoritmos e estruturas de dados. 3.ed. Pearson-Makron Books.2005.
  - Vetores: pag. 69
  - Estruturas: pag: 85