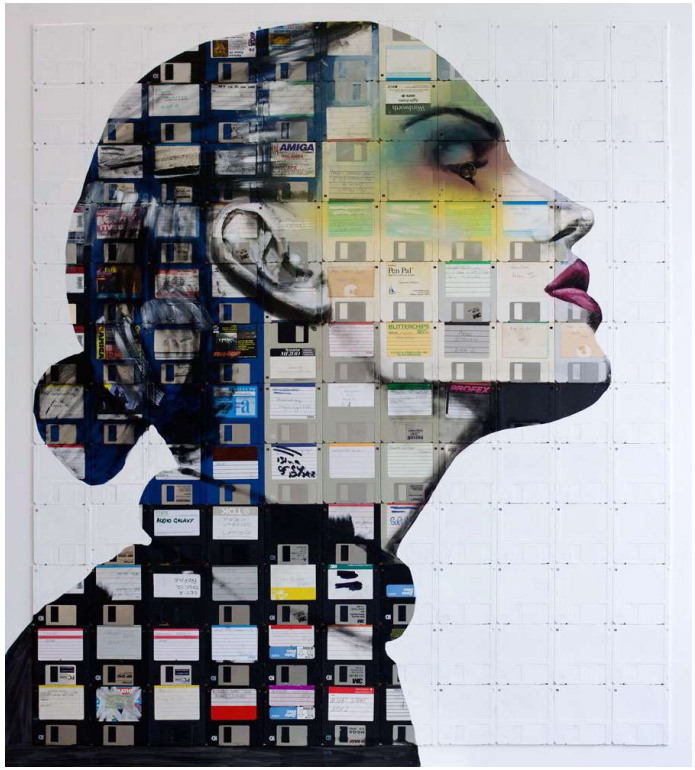
Algoritmos e Estruturas de Dados

Disciplina 301477



http://www.nickgentry.com/

Programa de Pós-graduação em
Computação Aplicada

Prof. Alexandre Zaghetto
http://alexandre.zaghetto.com
zaghetto@unb.br

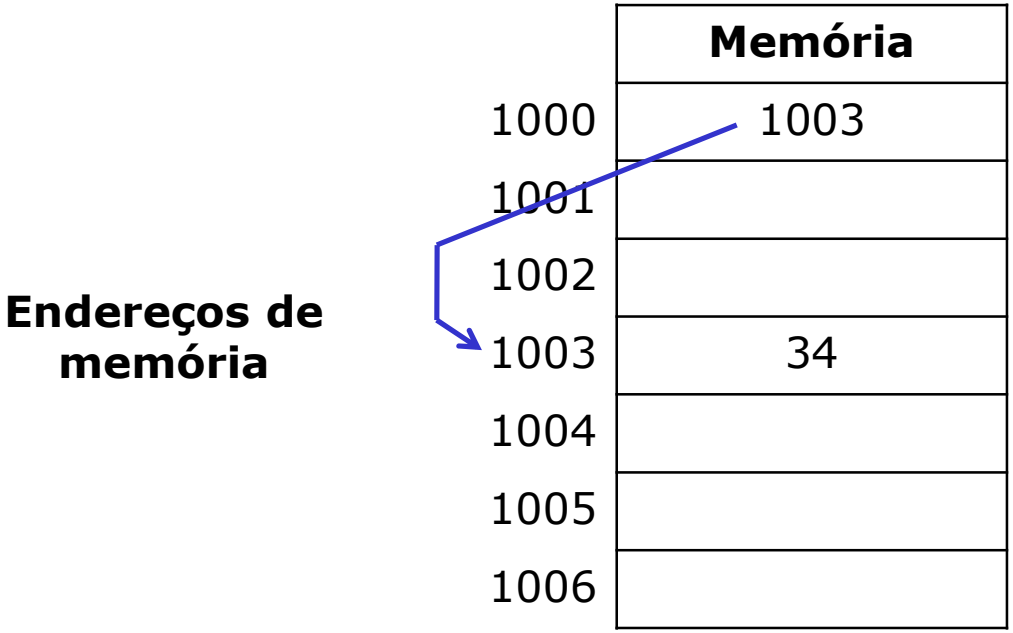
Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Módulo 10
Ponteiros

1. Ponteiros

- **Ponteiro** é uma variável que contém o endereço de uma outra variável.

- Daí o nome, pois ele **aponta** para outra variável.



1. Ponteiros

- Alguns usos:

  ✓ Manipular vetores e matrizes, incluindo strings.

  ✓ Modificar os argumentos (variáveis, vetores, matrizes e *structs*) de funções (passagem por referência).

  ✓ Alocar e desalocar memória dinamicamente.

  ✓ Passar para uma função o endereço de outra função.

  ✓ Criar estruturas de dados complexas.

1. Ponteiros

- Declaração de variáveis ponteiros:

```
tipo *nome
```

- Operadores de Ponteiros
  - ✓ Existem dois operadores especiais para ponteiros:
    - ✓ &
    - ✓ *

1. Ponteiros

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

int main (int argc, char *argv[])
{

    int x;

    x = 15;

    printf("CONTEUDO de X = %d \n", x);
    printf("ENDERECO de X = %d \n", &x);

    return 0;
}
```

& ➔ Pode ser lido como "o endereço de...".

1. Ponteiros

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

int main (int argc, char *argv[])
{

    int *p, x;

    x = 15;
    p = &x;

    printf("%d \n", p);
    printf("%d \n", *p);

    return 0;
}
```

* ➔ Pode ser lido como "o valor que está no endereço armazenado em..."

1. Ponteiros

- Aritmética de ponteiros:
  - ✓ Existem duas operações possíveis com ponteiros:
    - ✓ Adição; e
    - ✓ Subtração.

1. Ponteiros

- Aritmética de ponteiros:

  ✓ Consideremos **p1** um ponteiro para um inteiro com valor atual 1000. Assuma, também, que os inteiros são de 4 bytes.

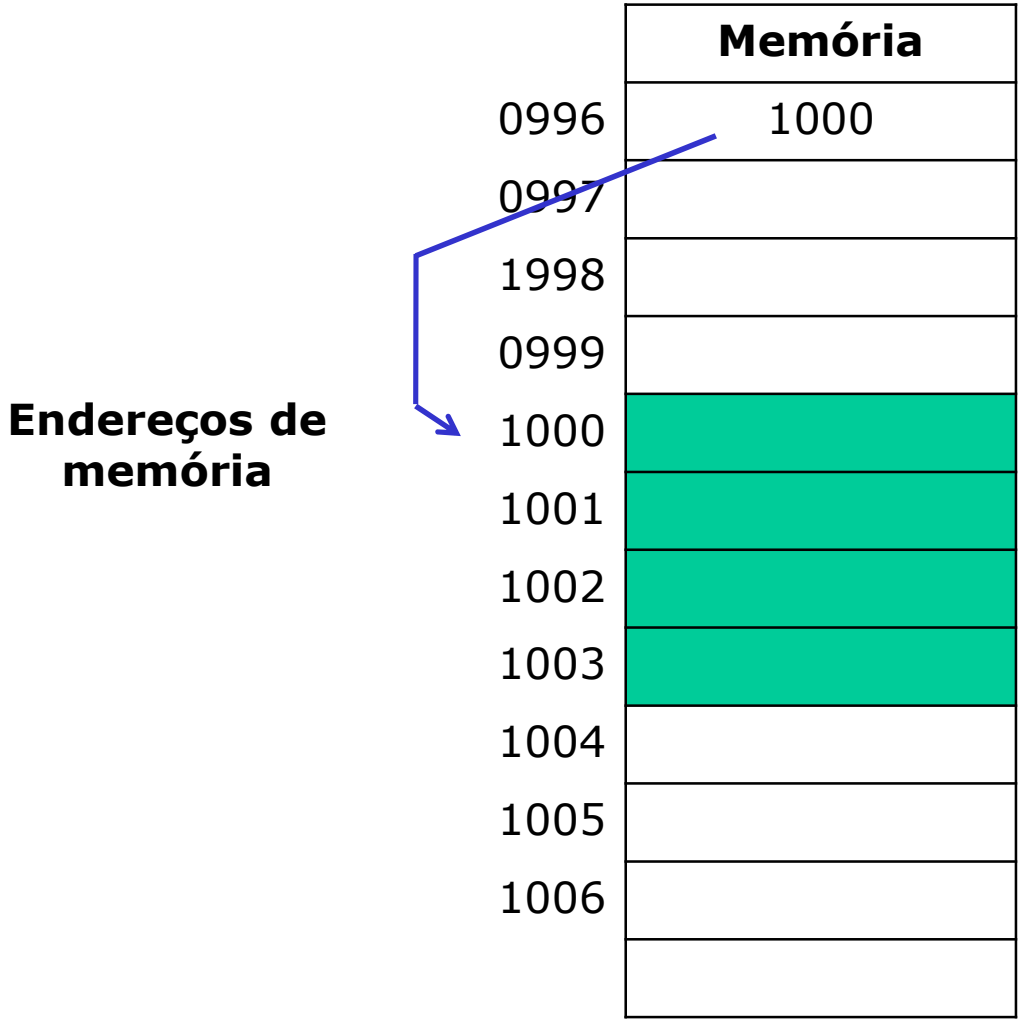
  ✓ Após a expressão p1++, p1 contém 1004.

  ✓ Cada vez que **p1 é** incrementado, ele aponta para o próximo inteiro.

  ✓ O mesmo é verdade nos decrementos.

  ✓ Ou seja, ponteiros incrementam ou decrementam **pelo tamanho do tipo de dado** que eles apontam.

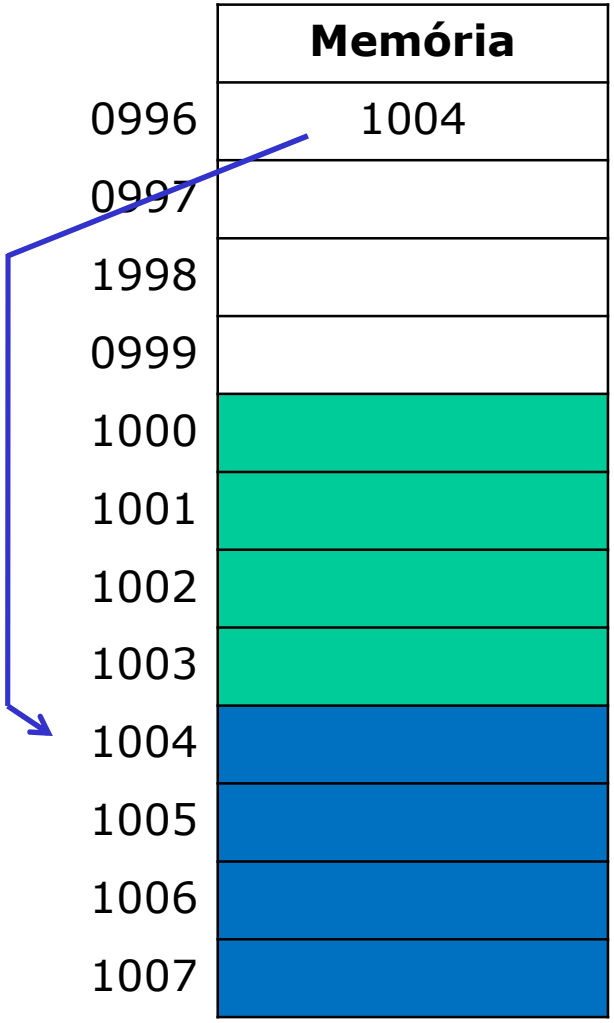
1. Ponteiros



p1

- p1 é um ponteiro para int
- int ocupa 4bytes, ou seja 4 posições de memória.

1. Ponteiros



Endereços de memória

• após p1++, o endereço armazenado em p1 passa a ser 1004, ou seja, aponta para o "próximo" inteiro.

p1

• p1 é um ponteiro para int

• int ocupa 4bytes, ou seja 4 posições de memória.

1. Ponteiros

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

int main (int argc, char *argv[])
{
    int *p1, x = 10;

    p1 = &x;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %d \n", x);

    p1++;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %d \n", x);


    return 0;
}
```

1. Ponteiros

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

int main (int argc, char *argv[])
{
    char *p1, x = 'a';

    p1 = &x;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %c \n", x);

    p1++;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %c \n", x);


    return 0;
}
```

1. Ponteiros

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

int main (int argc, char *argv[])
{
    double x = 1.23212345, *p1;

    p1 = &x;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %lf \n", x);

    p1 = p1 + 2;

    printf("p1: %d \n", p1);
    printf("&x: %d \n", &x);
    printf("x: %lf \n", x);


    return 0;
}
```

2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int main (int argc, char *argv[])
{
   int i, x[MAX] = {0,1,2,3};

   printf("Endereco\t  Conteudo\t \n");

   printf("Notacao de vetor:\n");
   for (i = 0; i<MAX; i++)
      printf("%d\t\t  %d\t \n", &x[i], x[i]);

   return 0;
}
```

2. Ponteiros e Vetores

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4

int main (int argc, char *argv[])
{
   int i, x[MAX] = {0,1,2,3};

   printf("Endereco\t  Conteudo\t \n");

   printf("Notacao de ponteiro:\n");
   for (i = 0; i<MAX; i++)
      printf("%d\t\t  %d\t \n", x+i, *(x+i));

   return 0;
}
```

3. Passagem de Parâmetros por Referência

- "Retornando" vários valores:

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

int retornavarios (float *, int *);

int main (int argc, char *argv[])
{
   float y = 5.0;
   int x = 5, sucesso;

   printf("y = %.2f - x = %d \n", y, x);

   sucesso = retornavarios(&y, &x);

   printf("y = %.2f - x = %d \n", y, x);
   printf("sucesso = %d \n", sucesso);
```

3. Passagem de Parâmetros por Referência

```
    return 0;
}

int retornavarios( float *n1, int *n2){

  *n1 = *n1/2;

  *n2 = *n2%2;

  return 0;
}
```

3. Passagem de Parâmetros por Referência

- Passando vetores por referência:

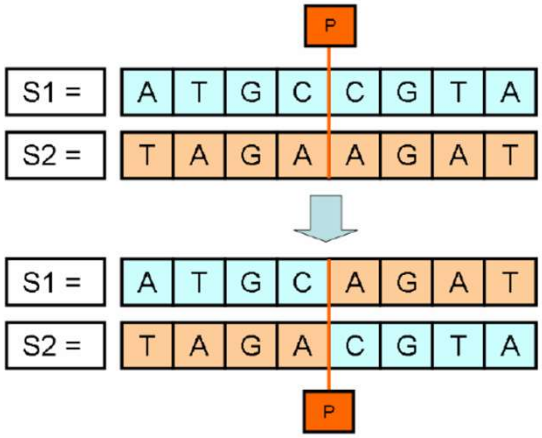
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

int retornavetor (float *, int);

int main (int argc, char *argv[])
{
   float x[MAX] = {0,0,0,0,0,0,0,0,0,0};
   int i, sucesso;

   printf("Vetor antes de chamar a funcao\n");

   for (i = 0; i<MAX; i++)
    printf("%.2f \n", x[i]);

   sucesso = retornavetor(x, MAX);
```

3. Passagem de Parâmetros por Referência

```
    printf("Vetor depois de chamar a funcao\n");
    for (i = 0; i<MAX; i++)
     printf("%.2f \n", x[i]);


    return 0;
}

int retornavetor( float *vet, int N){

  int i;

  for (i = 0; i<N; i++)
   *(vet+i) = i;

  return 0;
}
```

3. Passagem de Parâmetros por Referência

Exemplo: Um operador de *crossover* pode ser aplicado a duas strings s1 e s2 e consiste em se sortear aleatoriamente um ponto de s1 e s2 e, escolhido este ponto, é realizada a troca de informações de s1 e s2 tal como mostrado no esquema a seguir.



Escreva uma função que recebe duas strings s1 e s2 e realiza a operação de *crossover.* Escreva também um programa principal que utiliza a função proposta.

4. Ponteiros e Matrizes

- Considere o código a seguir:

```
#include <stdio.h>
#include <stdlib.h>
#define LIN 3
#define COL 3

int main(){

    int m[LIN][COL];
    int i, j;

    for (i=0; i<LIN; i++){
        for (j=0; j<COL; j++){
            printf("Elemento %d %d = ", i, j);
            scanf("%d", &m[i][j]);
        }
    }
```

4. Ponteiros e Matrizes

```
// Notação de matriz
 for (i=0; i<LIN; i++){
    for (j=0; j<COL; j++){
       printf("%d\t\t  %d\t \n", &m[i][j], m[i][j]);
    }
 }


   return 0;
}
```

• Execute o programa e note que os elementos da matriz são organizados em posições consecutivas da memória.

4. Ponteiros e Matrizes

- Agora veja a notação de ponteiro:

```
#include <stdio.h>
#include <stdlib.h>
#define LIN 3
#define COL 3

int main(){

    int m[LIN][COL], *p;
    int i, j;

   for (i=0; i<LIN; i++){
        for (j=0; j<COL; j++){
            printf("Elemento %d %d = ", i, j);
            scanf("%d", &m[i][j]);
        }
    }
```

4. Ponteiros e Matrizes

```
p = &m[0][0];

// Notação de ponteiro
 for (i=0; i<LIN; i++){
   for (j=0; j<COL; j++){
     printf("%d\t\t  %d\t \n", p+i*COL+j, *(p+i*COL+j));
   }
 }


   return 0;
}
```

- Observe que o efeito é o mesmo.

5. Ponteiros e Structs

- Passando structs como parâmetro de funções.

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

// Estrutura
struct dados_aluno{
    char nome[80];
    float media;
};

// Protótipo da função que recebe estruturas
void imprime_struct( struct dados_aluno);
```

5. Ponteiros e Structs

```
int main (int argc, char *argv[])
{
    struct dados_aluno aluno;

    strcpy(aluno.nome,"Alexandre Zaghetto");
    aluno.media = 9.5;

    imprime_struct(aluno);


    return 0;
}

// Imprime a estrutura
void imprime_struct( struct dados_aluno parm){
     printf ("%s \n", parm.nome);
     printf ("%f \n", parm.media);
}
```

5. Ponteiros e Structs

- Ponteiro para struct.

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

// Estrutura
struct dados_aluno{
    char nome[80];
    float media;
};

// Imprime a estrutura
void imprime_struct(struct dados_aluno *);

// Altera a estrutura
void altera_struct(struct dados_aluno *);
```

5. Ponteiros e Structs

```
int main (int argc, char *argv[])
{
    struct dados_aluno aluno, *p_aluno;

    p_aluno = &aluno;

    strcpy((*p_aluno).nome,"Alexandre Zaghetto");
    (*p_aluno).media = 9.5;

    printf("%s \n", (*p_aluno).nome);
    printf("%f \n", (*p_aluno).media);


    return 0;
}
```

5. Ponteiros e Structs

```
int main (int argc, char *argv[])
{
    struct dados_aluno aluno, *p_aluno;

    p_aluno = &aluno;

    strcpy (p_aluno->nome,"Alexandre Zaghetto");
    p_aluno->media = 9.5;

    printf("%s \n", p_aluno->nome);
    printf("%f \n", p_aluno->media);


    return 0;
}
```

5. Ponteiros e Structs

```
int main (int argc, char *argv[])
{
    struct dados_aluno aluno, *p_aluno;

    p_aluno = &aluno;

    strcpy(p_aluno->nome,"Alexandre Zaghetto");
    p_aluno->media = 9.5;

    imprime_struct(p_aluno);

    altera_struct(p_aluno);

    imprime_struct(p_aluno);


    return 0;
}
```

5. Ponteiros e Structs

```
// Imprime a estrutura
void imprime_struct(struct dados_aluno *parm){
     printf ("%s \n", parm->nome);
     printf ("%f \n", parm->media);
}

// Altera a estrutura
void altera_struct(struct dados_aluno *parm){
     strcpy (parm->nome, "Zaghetto Alexandre");
     parm->media = 5.9;
}
```

6. Alocação Dinâmica de Memória

• A alocação dinâmica permite ao programador alocar memória para variáveis enquanto o programa está sendo executado.

• É possível criar um vetor ou matriz cujo tamanho somente será definido em tempo de execução.

• A linguagem C define 4 funções para alocação dinâmica de memória, disponíveis na biblioteca stdlib.h:

- ✓ *malloc*(): aloca memória;
- ✓ *calloc():* aloca memória;
- ✓ *realloc():* realoca memória*; e*
- ✓ *free*(): libera memória alocada.

6. Alocação Dinâmica de Memória

- A função *malloc*() possui o seguinte protótipo:

```
void *malloc (size_t size);
```

- A função *malloc*() lê a quantidade **size** de *bytes* a alocar, reserva a memória correspondente e retorna o endereço do primeiro *byte* alocado.

- size_t é um tipo unsigned int.

- A função devolve um ponteiro do tipo *void*, o que significa que você pode atribui-lo a qualquer tipo de ponteiro.

- Se não houver memória disponível para alocar, a função retorna um ponteiro nulo (NULL).

6. Alocação Dinâmica de Memória

- A função *calloc*() possui o seguinte protótipo:

```
void *calloc (size_t num, size_t size);
```

- A função *calloc*() lê a quantidade **num** de elementos a alocar, cada qual com um tamanho de **size** bytes, reserva (**num*size**) bytes, inicializa o espaço alocado com 0 e retorna o endereço do primeiro *byte* alocado.

- Se não houver memória disponível para alocar, a função retorna um ponteiro nulo (NULL).

6. Alocação Dinâmica de Memória

- A função *realloc*() possui o seguinte protótipo:

```
void *realloc (void *ptr, size_t size);
```

- A função *realloc*() realoca (expande ou contrai) um espaço de memória previamente alocado, apontado por *ptr*. O novo tamanho passa a ser **size** *bytes***.**

- Se não houver memória disponível para realocar, a função retorna um ponteiro nulo (NULL).

6. Alocação Dinâmica de Memória

- A função *free()* possui o seguinte protótipo:

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

- Desaloca um bloco de memória apontado por *ptr,* previamente alocado por meio das funções *malloc(), calloc() ou realloc().*

6. Alocação Dinâmica de Memória

- Alocação de vetores:

  ✓ Exemplo de alocação para 30 valores do tipo double, utilizando *malloc()*:

```
double *p;

p = (double *) malloc(30 * sizeof(double));
```

  ✓ A operação realizada com o (double *) é chamada de *casting*. Ela garante que o ponteiro retornado pela função malloc() seja um ponteiro para double.

6. Alocação Dinâmica de Memória

- Alocação de vetores:

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

int main(){

   double *p;
   int N, numero, i;

   printf("Qual tamanho do vetor: ");
   scanf("%d", &N);

   p = (double *) malloc(N*sizeof (double));

   if (p == NULL){
        printf("Alocacao falhou. Finalizado.\n");
        exit(1);
   }
```

6. Alocação Dinâmica de Memória

```
    for(i=0; i < N; i++)
    {
        printf("Digite o valor %d do vetor: ", i);
        scanf("%lf", p+i);
        // scanf("%lf", &p[i]);
    }

    for(i=0; i < N; i++)
    {
        printf("%.2lf \n", *(p+i));
        //printf("%.2lf \n", p[i]);
    }

  free(p);


  return 0;
}
```

6. Alocação Dinâmica de Memória

- Alocação de memória com **calloc** e relocação com ***realloc***:

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

int main(){

   double *p;
   int N, M, numero, i;

   printf("Qual tamanho do vetor: ");
   scanf("%d", &N);

   p = (double *) calloc(N, sizeof (double));

   if (p == NULL){
        printf("Alocacao falhou. Finalizado.\n");
        exit(1);
   }
```

6. Alocação Dinâmica de Memória

```
for(i=0; i < N; i++){
    printf("%.2lf \n", *(p+i));
} // calloc preenche o espaço alocado com 0's.

for(i=0; i < N; i++){
    printf("Digite o valor %d do vetor: ", i);
    scanf("%lf", p+i);
}

printf("Qual o novo tamanho do vetor: ");
scanf("%d", &M);

p = (double *) realloc(p, M*sizeof (double));

if (p == NULL){
     printf("Realocacao falhou. Finalizado.\n");
     exit(1);
}
```

6. Alocação Dinâmica de Memória

```
    for(i=N; i < M; i++)
    {
        printf("Digite o valor %d do vetor: ", i);
        scanf("%lf", p+i);
    }

    for(i=0; i < M; i++)
    {
        printf("%.2lf \n", *(p+i));
    }

    free(p);


    return 0;
}
```

6. Alocação Dinâmica de Memória

- Alocação de matrizes:

  ➢ A alocação dinâmica de matrizes é realizada por meio das funções de manipulação de memória já apresentadas.

  ➢ Pode ser feitas de duas maneiras:

    1. Utilizando um único ponteiro e "entendendo" os valores lidos como sendo elementos de uma matriz.

    2. Utilizando ponteiro para ponteiro.

6. Alocação Dinâmica de Memória

- Alocação de matrizes (utilizando um único ponteiro):

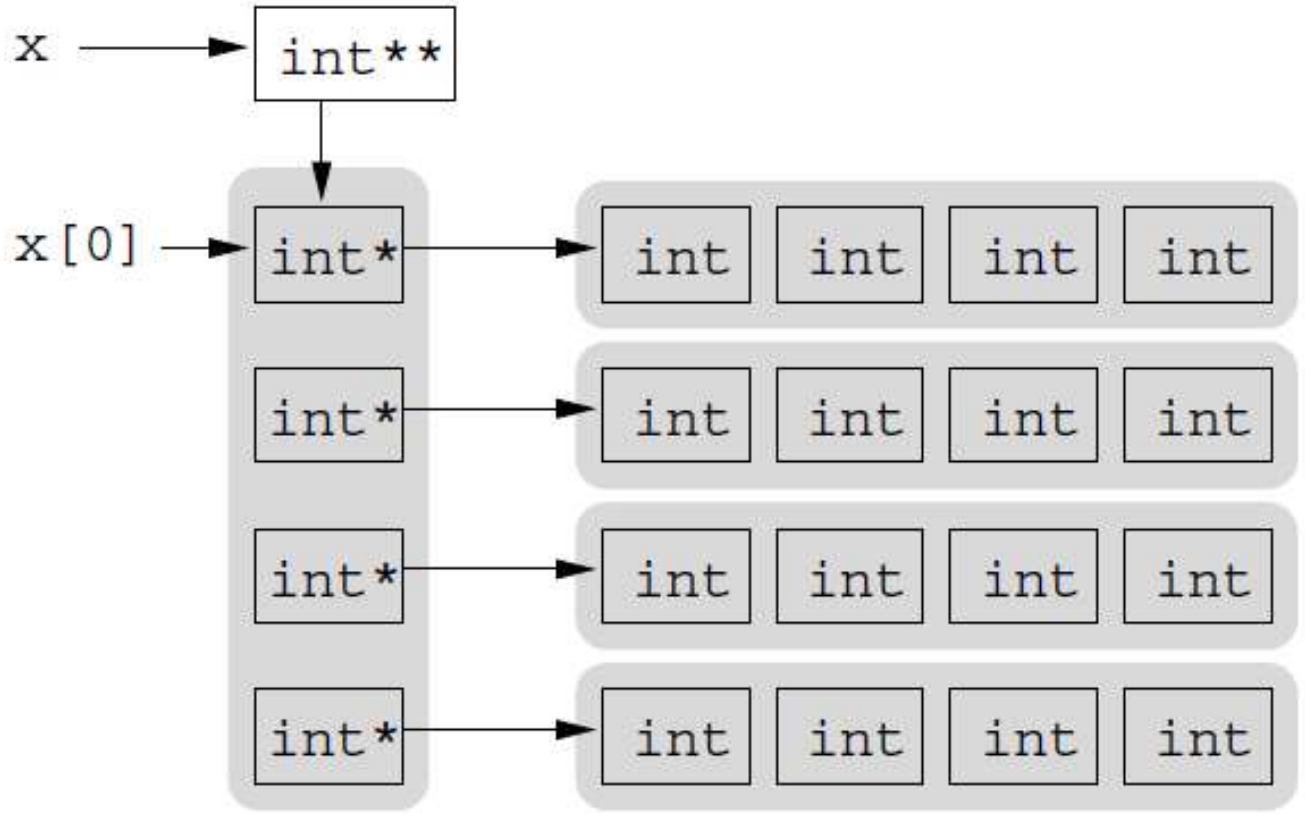
```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{

  int i,j, *mat;
  int Nlin, Ncol;

  printf("Digite o número de linhas da matriz:");
  scanf("%d", &Nlin);
  printf("Digite o número de colunas da matriz:");
  scanf("%d", &Ncol);

  mat = (int*) malloc(Nlin*Ncol*sizeof(int));
```

6. Alocação Dinâmica de Memória

```
    for (i=0;i<Nlin;i++){
      for(j=0;j<Ncol;j++)
      {
         printf("Digite o valor [%d][%d] da matriz:",i,j);
         scanf("%d", mat+(i*Ncol)+j);
      }
    }

    for (i=0;i<Nlin;i++){
      for (j=0;j<Ncol;j++)
      {
         printf("MAT[%d][%d]: %d \n",i,j, *(mat+(i*Ncol)+j));
      }
    }

     free(mat);

     return 0;
}
```

6. Alocação Dinâmica de Memória

- Alocação de matrizes (utilizando ponteiro para ponteiro):



6. Alocação Dinâmica de Memória

- Alocação de matrizes (utilizando ponteiro para ponteiro):

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

int main (int argc, char *argv[])
{
  int i,j, **mat;
  int Nlin, Ncol;

  printf("Digite o número de linhas da matriz:");
  scanf("%d", &Nlin);
  printf("Digite o número de colunas da matriz:");
  scanf("%d", &Ncol);

  mat = (int**)malloc(Nlin*sizeof(int *));

  for(i = 0; i<Nlin; i++)
    *(mat+i) = (int*)malloc(Ncol*sizeof(int));
   // mat[i] = (int*)malloc(Ncol*sizeof(int));
```

6. Alocação Dinâmica de Memória

```
for (i=0;i<Nlin;i++){
  for (j=0;j<Ncol;j++){
     printf("Digite o valor [%d][%d] da matriz:",i,j);
     scanf("%d", *(mat+i)+j);
     //scanf("%d",&mat[i][j]);
  }
}

for (i=0;i<Nlin;i++){
  for (j=0;j<Ncol;j++){
    printf("MAT[%d][%d]: %d \n",i,j, *(*(mat+i)+j));
   //printf("MAT[%d][%d]: %d \n",i,j, mat[i][j]);
  }
}
```

6. Alocação Dinâmica de Memória

```
    for (i=0;i<Nlin;i++)
        free(*(mat+i));

    //for (i=0;i<Nlin;i++)
    //    free(mat[i]);

    free(mat);


     return 0;
}
```

6. Alocação Dinâmica de Memória

- Passando *matrizes* por referência:

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

void imprimematriz(int **, int, int);

int main(int argc, char *argv[])
{

   int L = 4, C = 3, **M;
   int i, j;

   M = (int **)malloc(L*sizeof(int *));

   for(i = 0; i< L; i++)
        *(M+i) = (int *)malloc(C*sizeof(int));
```

6. Alocação Dinâmica de Memória

```
 for(i = 0; i<L; i++)
   for(j = 0; j<C; j++)
         M[i][j] = i*j;

  imprimematriz(M, L, C);


  return 0;
}

void imprimematriz(int **M, int L, int C){

  int i, j;

  for(i = 0; i<L; i++){
   for(j = 0; j<C; j++) printf("%d ", *(*(M+i)+j));
   printf("\n");
   }
}
```

"Ora, está longe de ser óbvio, de um ponto de vista lógico, haver justificativa no inferir enunciados universais de enunciados singulares, independentemente de quão numerosos sejam esses; com efeito, qualquer conclusão colhida desse modo sempre pode revelar-se falsa: independentemente de quantos casos de cisnes brancos possamos observar, isso não justifica a conclusão de que *todos* os cisnes são brancos." Karl Popper