Programacao em computadores II

Alocação dinâmica de memória

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Prof. Dr. Fábio Rodrigues de la Rocha

Introdução: Alocação estática x alocação dinâmica de memória

- Toda a vez que uma variável é declarada, esta é alocada **estaticamente** na memória, isto é, o compilador armazena no arquivo binário produzido que existe uma variável/vetor de determinado tamanho. Quando chamamos o programa para executá-lo, o mesmo é carregado para a memória RAM e neste momento a memória para a tal variável é alocada.
- Temos que na alocação estática a memória é alocada em tempo de **carga** do programa na RAM. Uma vez carregado, todas as variáveis já estão definidas com seus tamanhos e estes não podem ser mudados.
- Ocorre que as vezes não sabemos quantas posições de memória serão necessárias. Por exemplo, deseja-se escrever um programa em C que solicita ao usuário quantos numeros serão informados. Depois, o programa lê a tal quantidade de números e armazena em cada posição de um vetor. Pergunta: qual o tamanho do vetor ?

Introdução: Alocação estática x alocação dinâmica de memória

- Podemos imaginar que o tamanho do vetor seja 1000 e declarar int vetor[1000]
- Caso o número de elementos a ler durante a execução do programa seja bem menor haverá grande desperdício de memória;
- Por outro lado, caso seja necessária um vetor de mais de 1000 posições o programa não funcionará;

Introdução: Alocação estática x alocação dinâmica de memória

- Chamamos de alocação dinâmica de memória quando o programador deve invocar uma função para obter memória do sistema operacional. Essa memória pode ser solicitada a qualquer momento. Ou seja, um programa pode perguntar ao usuário quantos elementos ele deseja informar. Com posse desse valor, solicita-se exatamente esta quantidade de memória para o sistema operacional. Dessa forma, o tamanho do vetor será exatamente do tamanho necessário para armazenar os elementos do usuário.

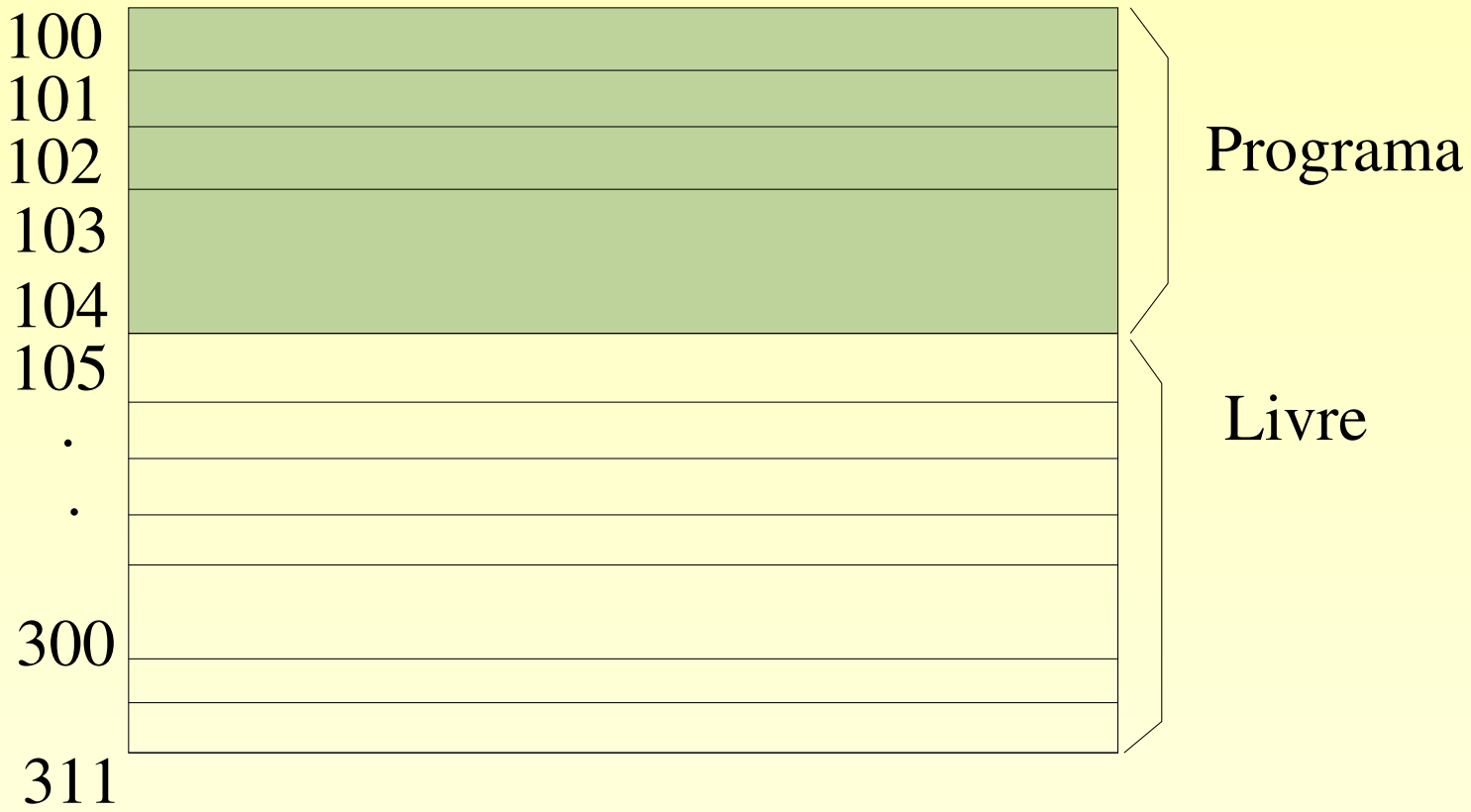
Alocação dinâmica de memória

Definição:

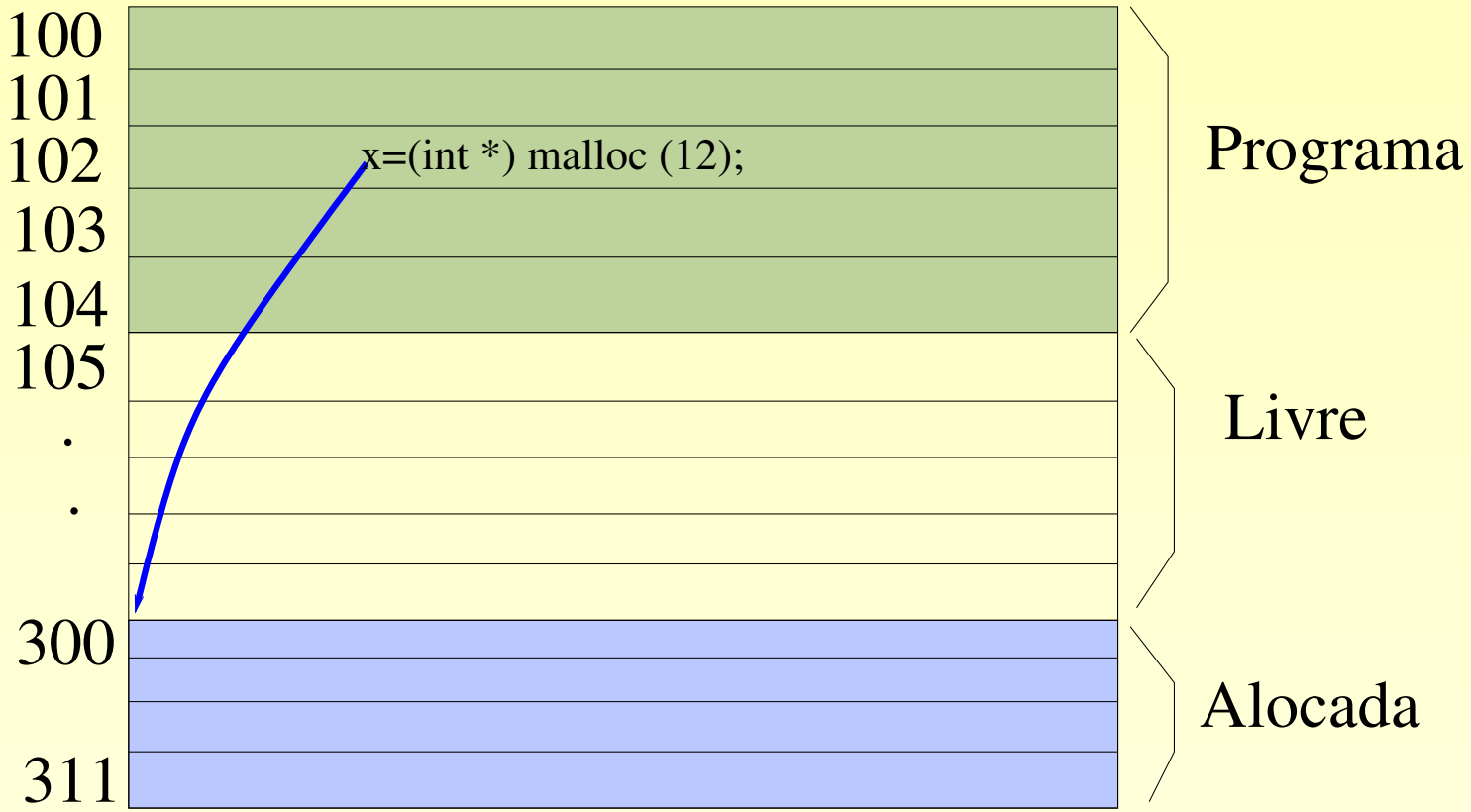
Na linguagem C existem algumas funções para solicitar memória para o sistema operacional. A mais conhecida é `malloc` - cujo nome deriva de **m***emory* **alloc***ate*.

```
1 int *x;
2     x = (int *) malloc (12);
3     // Solicitou-se 12 bytes para o SO
4     // Se existe memoria disponivel,
5     // o SO retornar o endereco de inicio da memoria
6     // cada inteiro ocupa 4 bytes, logo temos 3 inteiros.
7     x[0]=234;
8     x[1]=123;
9     x[2]=450;
```

Memória internamente



Memória internamente



Alocação dinâmica de memória

No código do slide anterior, digamos que o SO tenha encontrado 12 bytes de memória livre iniciando no endereço 300. Ou seja, seriam os endereços 300,...,311. Nesta situação a função `malloc` retornará o endereço 300. Precisamos de uma variável ponteiro para armazenar este endereço. No programa em questão, temos `int *x` que é um ponteiro para inteiro.

Alocação dinâmica de memória

A função `malloc` é feita para retornar um endereço qualquer, não somente para variáveis inteiras, mas para floats, double, ou variavéis que o programador tenha definido (`typedef` e `struct`). Para transformar o retorno da função `malloc()` para ser compatível com a variável `x` usamos o operador **cast** ou **operador de mudança de tipo**. `x = (int *) malloc (12);`

Desalocando memória

Memória que foi alocada por `malloc()` e não é mais utilizada pode ser liberada pelo programador C para permitir que o SO faça algum uso desta. Internamente, a cada chamada `malloc()` o sistema contabiliza o requisito de memória e insere numa tabela de acesso restrito.

```
1 int *x, *z, *k;
2
3 x = (int *) malloc (12);
4 // vamos assumir que x=20
5 z = (int *) malloc (120);
6 // vamos assumir que y=328
7 k = (int *) malloc (234);
8 //vamos assumir que k=4000
```

End. Inicial	qtd bytes
20	12
328	120
4000	4000

Alocação dinâmica de memória

Desalocando memória

Para desalocar a memória, utilizamos `free(endereco_inicial)`.

```
1 int *x, *z, *k;
2
3 x = (int *) malloc (12);
4 // vamos assumir que x=20
5 z = (int *) malloc (120);
6 // vamos assumir que y=328
7 k = (int *) malloc (234);
8 //vamos assumir que k=4000
9 free(z); //libera 120 bytes
```

Alocação dinâmica de memória

Descobrindo o tamanho de alguma variável

Usando a instrução `sizeof()`.

```c
#include <stdio.h>
int main (void)
{
   // Tipos de variaveis tipicos em C
   printf("Tamanho de um char   =%d byte(s)\n",sizeof(char));
   printf("Tamanho de um int    =%d byte(s)\n",sizeof(int));
   printf("Tamanho de um short int =%d byte(s)\n",sizeof(short int));
   printf("Tamanho de um long   =%d byte(s)\n",sizeof(long));
   printf("Tamanho de um long long =%d byte(s)\n",sizeof(long long));
   printf("Tamanho de um float  =%d byte(s)\n",sizeof(float));
   printf("Tamanho de um double =%d byte(s)\n",sizeof(double));

   // Em 1999 foram adicionados os novos tipos inteiros
   //int8_t: signed 8-bit
   //uint8_t: unsigned 8-bit
   //int16_t: signed 16-bit
   //uint16_t: unsigned 16-bit
   //int32_t: signed 32-bit
   //uint32_t: unsigned 32-bit
   //int64_t: signed 64-bit
   //uint64_t: unsigned 64-bit
   return 0;
}
```

Alocando memória com calloc

Além do malloc(), existe uma outra função de alocação de memória chamada **calloc**. O nome calloc vem de **c**lean and **alloc**.
Como o nome diz é uma função que aloca um pedaço da memória e limpa esse pedaço (ou seja, preenche com zeros). A forma de invocar a função é um pouco diferente: `void *calloc(size_t nmemb, size_t size);`
Observe o exemplo do próximo slide. Faça o download e teste o exemplo.

Código:exemplo_calloc.c

Alocação dinâmica de memória

```c
#include <stdlib.h>
#include <stdio.h>
#define TAM 20
int main (void) {
    int *vetor1;  int *vetor2;  int x;

    // aloca memoria com malloc e mostra o conteudo
    // poderia acontecer que a memoria ja esteja zerada
    // entao, vamos escrever algo e desalocar
    // com sorte quando alocarmos novamente
    // a area de memoria sera a mesma
    // e tera o conteudo antigo ja gravado

    vetor1 = (int *) malloc (TAM*sizeof(int));
    for (x=0;x<TAM;x++) vetor1[x]=x;
    free(vetor1);
    vetor1 = (int *) malloc (TAM*sizeof(int));
    for (x=0;x<TAM;x++) printf("Vetor 1[%d]=%d\t",x,vetor1[x]);
    printf("\n\n");

    // aloca memoria com calloc e mostra o conteudo
    vetor2 = (int *) calloc (TAM,sizeof(int));
    for (x=0;x<TAM;x++) vetor2[x]=x;
    free(vetor2);
    vetor2 = (int *) calloc (TAM,sizeof(int));
    for (x=0;x<TAM;x++) printf("Vetor 2[%d]=%d\t",x,vetor2[x]);
    return 0;
}
```

Alocando memória com realloc

Já vimos que podemos alocar memória com `malloc()/calloc()` e desalocar com `free()`.
Agora digamos que após ter alocado memória com digamos malloc, percebe-se que o tamanho precisaria ser maior. Através da função realloc pode-se "trocar" o pedaço de memória que já está alocado por um novo pedaço de tamanho diferente (seja maior ou menor).

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

Código:exemplo_realloc.c

Alocação dinâmica de memória

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

int main (void) {
   int *vetor; int x, quantidade, quantidade2;

   printf("Entre com a quantidade de numeros para ler:");

   scanf("%d",&quantidade);
   vetor = (int *) malloc (quantidade*sizeof(int));
   // o tamanho do vetor eh exatamente o necessario
   for (x=0;x<quantidade;x++) scanf("%d",&vetor[x]);

   printf("Entre com a nova quantidade de numeros para ler:");
   scanf("%d",&quantidade2);
   if (quantidade != quantidade2) {
      // se agora precisamos ler uma quantidade diferente de elementos
      // podemos trocar o vetor que temos, por outro
      vetor = (int *) realloc (vetor,quantidade2);
      // O interessante  que os elementos que estavam no vetor, permanecero na nova area
      // de memoria
   }
   for (x=0;x<quantidade;x++) printf("Vetor[%d]=%d\n",x,vetor[x]);
   for (x=0;x<quantidade2;x++) scanf("%d",&vetor[x]);
   return 0;
}
```

Alocação dinâmica de memória

A Linguagem C permite a definição de novos tipos a partir de tipos primitivos;

- typedef char letra;
- typedef int largura;
- typedef float preco;

Considerando o exemplo anterior, seria interessante a criação de um tipo chamado vetor:
typedef int *vetor;

Código:exemplo_typedef.c

Alocação dinâmica de memória

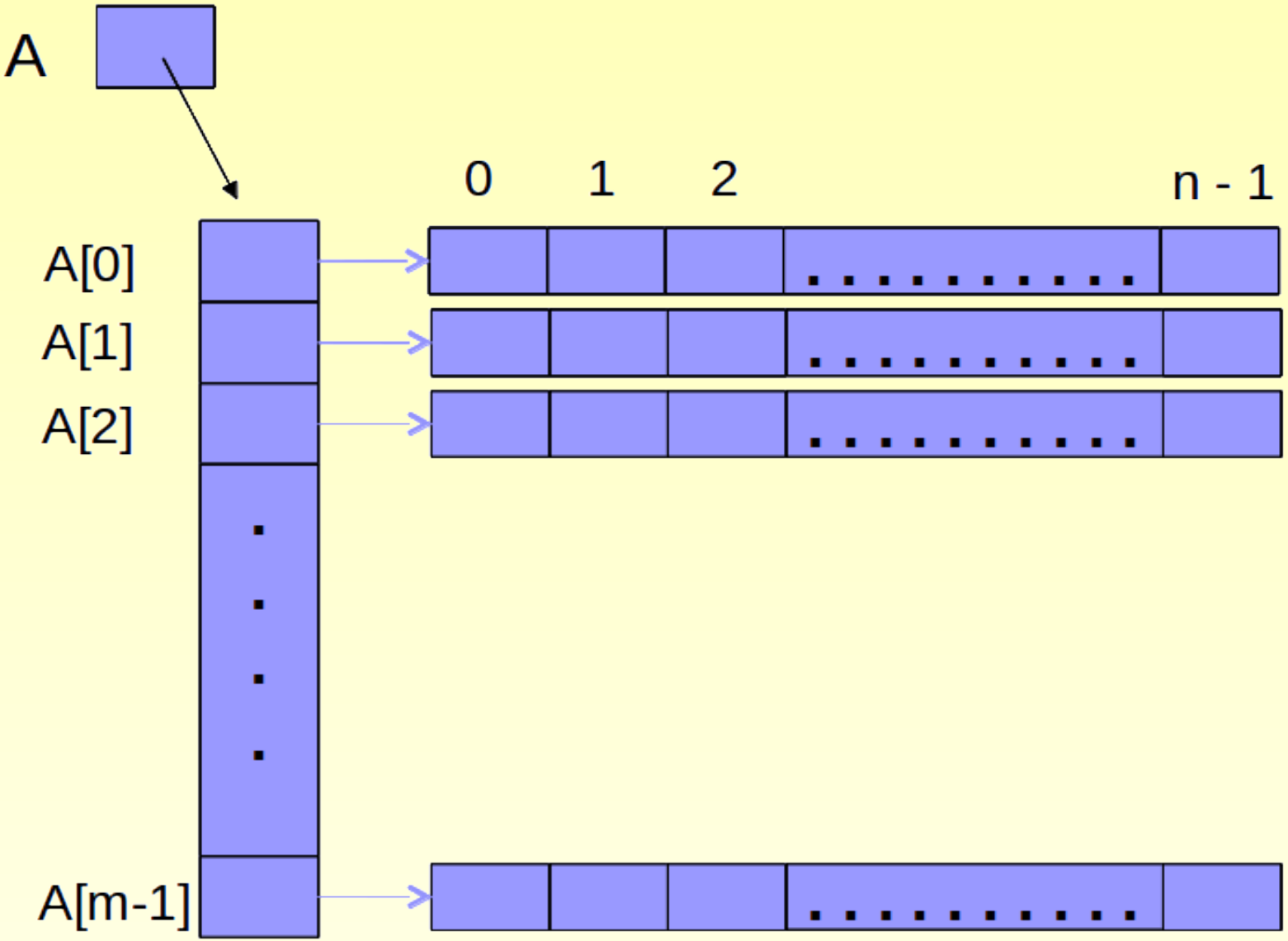
```
1
2  #include <stdio.h>
3  typedef int *vetor;
4
5  main() {
6      int m, i;
7      vetor A;
8      printf("Informe o tamanho do vetor: ");
9      scanf("%d",&m);
10     printf("\n");
11     A = (int *) malloc(m*sizeof(int));
12     for (i=0; i < m; i++) {
13         printf("Informe o %d. valor do vetor: ", i+1);
14         scanf("%d", &A[i]);
15     }
16     printf("\n");
17     for (i=0; i < m; i++) {
18         printf("\nO %d. valor do vetor : %d",i+1, A[i]);
19     }
20 }
```

Alocação dinâmica de memória: Matrizes

Matrizes podem ser entendidas como vetores de vetores de algum tipo;
Nesse sentido deve-se alocar um vetor que represente as linhas da matriz e para cada linha alocar um vetor que irá representar as colunas da linha em questão;
Utilizando typedef e considerando o exemplo anterior teríamos:

- typedef int *vetor; typedef vetor *matriz
- é a mesma coisa que: typedef int **matriz;

Alocação dinâmica de memória: Matrizes



Alocação dinâmica de memória: Matrizes

- No exemplo anterior, o tipo vetor é um ponteiro para inteiros e o tipo matriz é um ponteiro para o tipo vetor;
- Antes de utilizar uma matriz ("A" considerando a figura anterior) como variável indexada bidimensional deve-se, a partir da obtenção do número de linhas (m) e colunas (n), realizar a alocação de m bytes
  A = (vetor *) malloc(m * sizeof(vetor));
- em seguida deve-se alocar um conjunto de n * sizeof(int) bytes
  for (i=0; i < m; i++) A[i] = (int *) malloc(n * sizeof(int));

Alocação dinâmica de memória: Matrizes

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  main() {
5      int m, n, i, j;
6      int **A;
7      printf("Informe o nmero de linhas da matriz: ");
8      scanf("%d",&m);
9      printf("Informe o nmero de colunas da matriz: ");
10     scanf("%d",&n);
11     printf("\n");
12     A = (int **) malloc(m*sizeof(int));
13     for (i=0; i < m; i++) {
14         A[i] = (int *) malloc(n * sizeof(int));
15     }
```

Alocação dinâmica de memória: Matrizes

```
1  //Preenche a matriz
2     for (i=0; i < m; i++) {
3         for (j=0; j < n; j++) {
4             printf("Informe o elemento da linha %d, coluna %d: ", i+1, j+1);
5             scanf("%d", &A[i][j]);
6         }
7     }
8
9     //Apresenta o conteda da matriz
10    printf("\n");
11    for (i=0; i < m; i++) {
12        for (j=0; j < n; j++) {
13            printf
14            ("\nO elemento da linha %d, coluna %d : %d", i+1, j+1, A[i][j]);
15        }
16    }
17
18    //Libera a matriz
19    for (i=0; i < m; i++)
20        free(A[i]);
21     free(A);
22 }
```

Código:exemplo_matriz.c

Exercícios

Exercício 1:

Considerando a alocação dinâmica de matrizes crie um programa com uma função que receba o tamanho de uma matriz (linha e coluna) e retorne uma matriz alocada dinamicamente. Lembre que uma matriz, quando alocada dinamicamente é um ponteiro de ponteiro.

Exercícios

Exercício 2:

Considerando o exercício anterior modifique o programa e a função que cria a matriz dinamicamente tendo como base o conceito de typedef discutido nessa unidade.

Exercícios

Exercício 3:

Faça um programa que preencha uma matriz M (2x2), calcule e mostre a matriz R, resultante da multiplicação dos elementos de M pelo seu maior elemento.
Obs: As matrizez M e R devem ser alocadas dinamicamente utilizando a função criada anteriormente.

Exercícios

Exercício 4:

Faça um programa que preencha uma matriz (5x3) com as notas de 5 alunos em três provas. O programa deverá mostrar um relatório com o número dos alunos (número da linha) e a prova (número da coluna) em que cada aluno obteve a menor nota. Ao final do relatório, deverá mostrar quantos alunos tiveram a menor nota em cada uma das provas, ou seja, na prova 1, na prova 2 e na prova 3.
Obs: A matriz e o vetor que armazena o acumulado das notas devem ser criados dinamicamente. Utilize a função de criação de matrizes e crie uma função para a alocação dinâmica de vetores.

Exercícios

Exercício 5:

Faça um programa que preencha uma matriz 3x5 com números inteiros e some cada uma das linhas, armazenando o resultado das somas em um vetor. A seguir, o programa deverá multiplicar cada elemento da matriz pela soma da linha correspondente e mostrar a matriz resultante.
Obs: Utilize as funções de criação de vetores e matrizes e trabalhe com o conceito de modularização. As funções de criação de matrizes e vetores devem ser transformadas em bibliotecas (arquivo .h) e importadas no programa principal.

Exercícios

Exercício 6:

Faça um programa que preencha uma matriz 4x5 e uma segunda matriz 5x2. O programa deverá, também, calcular e mostrar a matriz resultante do produto matricial das duas matrizes anteriores, que foi previamente armazenada em uma terceira matriz de ordem 4x2.
Obs: Utilize a biblioteca criada anteriormente e o conceito de modularização.

Exercícios

Exercício 7:

Escreva um programa em C que manipule um vetor de inteiros não nulos alocado dinamicamente. O programa recebe inteiros, através da entrada padrão, e os insere no vetor. A cada inteiro que é inserido a área de memória necessária para armazenar um inteiro é incrementada ao número de bytes necessários para armazenar o vetor. O vetor não ocupa memória inicialmente. Quando o usuário entrar com o inteiro -1, o programa será finalizado e o mesmo não pertencerá ao vetor. Após o processo de inserção o vetor deve ser impresso na saída padrão. Libere a memória utilizada antes do final do processamento.
Obs: Utilize a função realloc() assim como o conceito de modularização.

Exercícios

Exercício 8:

Elabore uma biblioteca com as funções de alocação dinâmica de vetores e matrizes, liberação de vetores e matrizes e realocação de vetores. Essas funções devem ser genéricas, ou seja, devem receber o tamanho de determinado tipo que se deseja armazenar e deve retornar sempre um ponteiro de void. Desse modo para se utilizar a estrutura de retorno (vetor ou matriz) será necessário a realização de typecast.
Crie ainda um programa que faça as chamadas dessas funções de modo que a biblioteca como um todo possa ser testada;