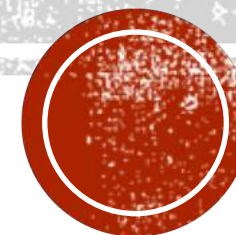


ALOCACÃO DINÂMICA DE MEMÓRIA

UNIDADES 15 E 16

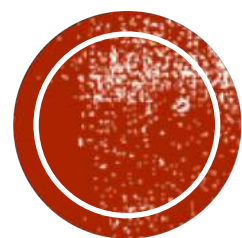
SI100 – Algoritmos e Programação de Computadores I
1º Semestre de 2018



Prof. Guilherme Palermo Coelho
guilherme@ft.unicamp.br

ROTEIRO

- Definições Gerais;
- Biblioteca *stdlib.h*;
- Vetores Locais e Funções;
- Alocação Dinâmica de Matrizes;
- Alocação Dinâmica de Estruturas;
- Exercícios;
- Referências.



DEFINIÇÕES GERAIS

ALOCAÇÃO DINÂMICA

- Nos exercícios que fizemos até agora enfrentamos as seguintes restrições:
 - Em programas que trabalhavam com vetores (matrizes), sempre era preciso saber o número **máximo** de elementos que poderiam estar presentes em um vetor;
 - Este número deve ser definido na codificação;
 - Pode levar a um desperdício de memória;
 - Impede o uso do programa em situações em que mais elementos são necessários.

ALOCAÇÃO DINÂMICA

- A linguagem C oferece mecanismos que permitem **requisitar espaços de memória** durante a execução do programa;
 - Isto é conhecido como *alocação dinâmica*;
- Programas que trabalham com vetores cujo número de elementos pode variar a cada execução:
 - Consulta-se o número de elementos que será necessário;
 - Aloca-se a quantidade de memória exata para aquela execução.

USO DA MEMÓRIA

- De maneira simplificada, pode-se dizer que existem três maneiras de se usar a memória para armazenar informações:
 - **Variáveis globais:** espaço em memória existe enquanto o programa estiver sendo executado;
 - **Variáveis locais:** espaço existe enquanto a função que declarou a variável estiver sendo executada;
 - **Alocação dinâmica:** programa requisita ao sistema, durante a execução, um espaço de determinado tamanho → existe até que seja explicitamente liberado (ou o programa termine).

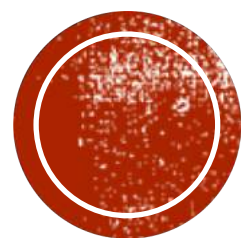
USO DA MEMÓRIA



- A cada chamada a funções o sistema reserva um espaço na pilha (variáveis locais e outras informações);
- O que sobra pode ser usado para alocação dinâmica.

Memória Livre

- Se a pilha tentar crescer além do espaço disponível, há um “estouro de memória” – *Stack Overflow*



BIBLIOTECA ***STD LIB***

FUNÇÕES DA BIBLIOTECA `STDLIB`

- Existem funções na biblioteca `stdlib.h` que permitem alocar e liberar memória dinamicamente:
 - `malloc()`: recebe como parâmetro o número de bytes que se deseja alocar e retorna o **endereço** inicial da área de memória alocada;
 - É utilizada com *ponteiros*;
 - Caso não haja espaço suficiente, é retornado um endereço nulo (NULL, também definido na `stdlib.h`);
 - Por definição, retorna um endereço genérico (`void*`) – recomenda-se uma conversão explícita de tipo (`cast`);

FUNÇÕES DA BIBLIOTECA `STDLIB`

- Existem funções na biblioteca `stdlib.h` que permitem alocar e liberar memória dinamicamente:
 - `malloc()`: recebe como parâmetro o número de bytes que se deseja alocar e retorna o **endereço** inicial da área de memória alocada;
 - Para se tornar independente de compiladores e arquiteturas, é usada em conjunto com o operador `sizeof()`;
 - `sizeof()`: retorna o tamanho, em bytes, do especificador de tipo (ou variável) passado como parâmetro;
 - **Ex.:** `sizeof(int)`;

FUNÇÕES DA BIBLIOTECA STDLIB

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

```
int main() {
    float* v;
    int n;

    scanf("%d", &n);

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

    if (v == NULL) {
        printf("Memoria insuficiente\n");
        return 1;
    }

    return 0;
}
```

Cast: converte para o tipo de v .

Cria o espaço correspondente a n floats.

Verifica e a criação se deu sem problemas.

Retorna um código de erro para o S.O.

FUNÇÕES DA BIBLIOTECA STDLIB

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

int main() {
    float* v;
    float med = 0.0;
    int n, i;

    scanf("%d", &n);

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

    if (v == NULL) {
        printf("Memoria insuficiente\n");
        return 1;
    }
    //...
```

```
        for (i=0; i<n; i++) {
            scanf("%f", &v[i]);
            med += v[i];
        }

        printf("A media é %.2f\n", med/n);

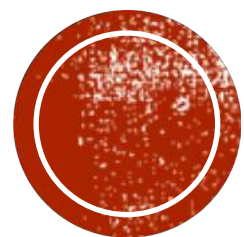
        free(v);

        return 0;
    }
```

Libera o espaço alocado a v.

FUNÇÕES DA BIBLIOTECA *STDLIB*

- Existem funções na biblioteca *stdlib.h* que permitem alocar e liberar memória dinamicamente:
 - *free()*: recebe como parâmetro o ponteiro da memória a ser liberada;
 - Só pode ser utilizada em espaços de memória alocados dinamicamente.



VETORES LOCAIS A FUNÇÕES

VETORES LOCAIS A FUNÇÕES

- A função abaixo está correta?

```
float* produto_vetorial(float* u, float* v)
{
    float p[3];

    p[0] = u[1]*v[2] - v[1]*u[2];
    p[1] = u[2]*v[0] - v[2]*u[0];
    p[2] = u[0]*v[1] - v[0]*u[1];

    return p;
}
```

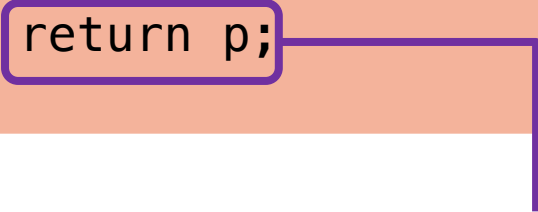
VETORES LOCAIS A FUNÇÕES

- A função abaixo está correta?

```
float* produto_vetorial(float* u, float* v)
{
    float p[3];

    p[0] = u[1]*v[2] - v[1]*u[2];
    p[1] = u[2]*v[0] - v[2]*u[0];
    p[2] = u[0]*v[1] - v[0]*u[1];

    return p;
}
```



ERRO: a variável *p* só existe dentro da função! Seu espaço é liberado ao final da execução!

VETORES LOCAIS A FUNÇÕES

- No exemplo anterior, o erro está no fato de retornarmos o valor de um endereço de memória que **não estará mais disponível** quando a função terminar sua execução:
 - A variável *p* é uma variável local;
 - A variável *p* está alocada na *pilha*!
- Possíveis soluções:
 - i) Retornar o vetor em um parâmetro cujo espaço foi alocado **fora** da função;
 - ii) Usar alocação dinâmica para definir *p*.

VETORES LOCAIS A FUNÇÕES

- Solução (i):

```
void produto_vetorial(float* u, float* v, float* p)
{
    p[0] = u[1]*v[2] - v[1]*u[2];
    p[1] = u[2]*v[0] - v[2]*u[0];
    p[2] = u[0]*v[1] - v[0]*u[1];
}
```

VETORES LOCAIS A FUNÇÕES

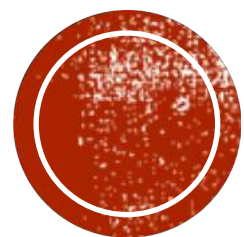
- Solução (ii):

```
float* produto_vetorial(float* u, float* v)
{
    float* p = (float*)malloc(3*sizeof(float));

    p[0] = u[1]*v[2] - v[1]*u[2];
    p[1] = u[2]*v[0] - v[2]*u[0];
    p[2] = u[0]*v[1] - v[0]*u[1];

    return p;
}
```

Cuidado: cada chamada à função alocará um vetor de 3 floats. Cabe à função que chama liberar o espaço utilizado.



ALOCACÃO DINÂMICA DE MATRIZES

ALOCAÇÃO DINÂMICA DE MATRIZES

- Matrizes declaradas estaticamente sofrem das mesmas limitações dos vetores → é preciso saber de antemão suas dimensões;
- Se elas só forem determinadas em tempo de execução, é necessário utilizar alocação dinâmica:
 - **Problema:** C só permite alocação dinâmica de vetores unidimensionais.

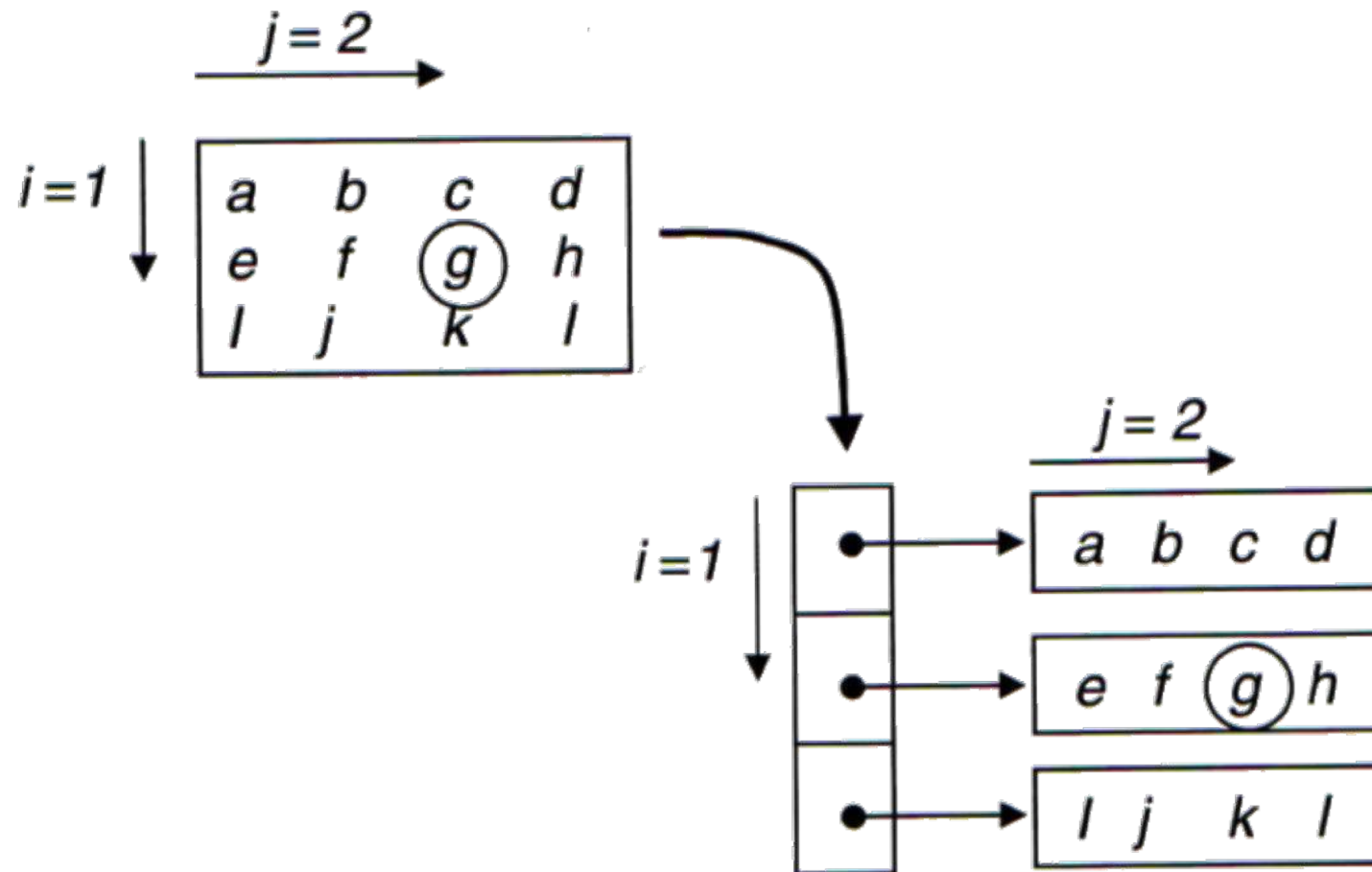
O que fazer?

ALOCAÇÃO DINÂMICA DE MATRIZES

- **Possível solução:** olhar para matrizes como *vetores unidimensionais* de vetores;
- Como para alocação dinâmica de vetores unidimensionais trabalhamos com ponteiros, no caso de matrizes teremos **vetores de ponteiros**:
 - Cada elemento armazena o endereço do primeiro elemento de cada linha (supondo matrizes bidimensionais).

ALOCAÇÃO DINÂMICA DE MATRIZES

- Caso 2D:



ALOCAÇÃO DINÂMICA DE MATRIZES

- A alocação dinâmica agora é mais elaborada:
 - Primeiro temos que alocar o ***vetor unidimensional de ponteiros***;
 - Depois aloca-se ***cada uma das linhas da matriz***:
 - Atribuindo seus endereços a cada elemento do vetor criado.

```
//Alocação dinâmica de uma matriz com m linhas e n colunas;  
int i;  
float** mat; //matriz representada por um vetor de ponteiros  
  
mat = (float**) malloc(m*sizeof(float*));  
  
for (i=0; i<m; i++)  
    mat[i] = (float*) malloc(n*sizeof(float));
```

ALOCAÇÃO DINÂMICA DE MATRIZES

- A vantagem desta forma de alocação é que o acesso a cada elemento se dá da mesma forma que uma matriz criada estaticamente: **mat[i][j]**;
- A **liberação** do espaço também deve seguir o mesmo processo:

```
...  
for (i=0; i<m; i++)  
    free(mat[i]);  
free(mat);
```

ALOCAÇÃO DINÂMICA DE MATRIZES

- **Exemplo:** transposição de matrizes

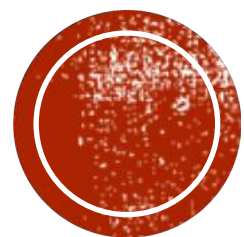
```
float** transposta(int m, int n, float** mat)
{
    int i, j;
    float** trp;

    trp = (float**) malloc(m*sizeof(float*));

    for (i=0; i<m; i++)
        trp[i] = (float*) malloc(n*sizeof(float));

    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            trp[j][i] = mat[i][j];

    return trp;
}
```

ALOCACÃO DINÂMICA DE ESTRUTURAS

PONTEIROS PARA ESTRUTURAS

- Como vimos na Unidade 12 do Curso (Tópico 8), estruturas (*structs*) são tipos de dados que agrupam campos de tipos (possivelmente) diferentes.

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

```
int main() {  
    struct ponto {  
        float x;  
        float y;  
    } p1;
```

```
    p1.x = 10.0;  
    p1.y = 5.0;
```

```
    printf("Coordenada x: %.2f – Coordenada y: %.2f\n", p1.x, p1.y);  
    return 0;
```

```
}
```

Coordenada x: 10.00 – Coordenada y: 5.00

PONTEIROS PARA ESTRUTURAS

- Da mesma maneira que declaramos ponteiros para outros tipos de dados, é possível declarar ***ponteiros para estruturas***:

```
struct ponto* pp;
```

- Nestes casos, o acesso aos membros pode se dar de maneira análoga o que já vimos anteriormente:

```
(*pp).x = 12.0;
```

→ Os parêntesis são **obrigatórios** → operador * tem precedência menor que .

PONTEIROS PARA ESTRUTURAS

- No entanto, o uso de estruturas (e de *ponteiros de estruturas*) é tão comum que existe outra forma de acesso aos membros de uma estrutura:

```
...  
struct ponto *pp;  
...  
pp->x = 12.0;  
...
```

- O operador “->” permite acessar os membros diretamente de um ponteiro para uma estrutura;
 - **CUIDADO**: não se esqueça de **alocar** espaço para a estrutura!

PONTEIROS PARA ESTRUTURAS

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

```
int main() {  
    struct ponto {  
        float x;  
        float y;  
    } p1;
```

```
    struct ponto* pp;  
    pp = &p1;
```

```
    p1.x = 10.0;  
    p1.y = 5.0;
```

```
    printf("Coordenada x: %.2f – Coordenada y: %.2f\n", pp->x, pp->y);  
    return 0;
```

```
}
```

Coordenada x: 10.00 – Coordenada y: 5.00

PASSAGEM DE ESTRUTURAS COMO PARÂMETROS

- Qual a diferença entre as funções abaixo?

```
void imprime(struct ponto p)
{
    printf("O ponto passado foi: (%.2f, %.2f)\n", p.x, p.y);
}
```

```
void imprime(struct ponto *pp)
{
    printf("O ponto passado foi: (%.2f, %.2f)\n", pp->x, pp->y);
}
```


PASSAGEM DE ESTRUTURAS COMO PARÂMETROS

- A impressão na tela, das duas funções, é idêntica;
- O seu funcionamento é bem diferente:
 - Na primeira, há uma ***passagem de parâmetro por valor***, ou seja, **toda a estrutura** é copiada para o parâmetro da função (dobrando o espaço de memória utilizado pela função);
 - Estruturas podem ser grandes!
 - Não é possível alterar o conteúdo da estrutura dentro da função;
 - Na segunda, ***apenas o ponteiro*** (geralmente 4 bytes) é copiado para o parâmetro (***passagem por referência***);
 - É possível alterar o conteúdo da estrutura dentro da função.

PASSAGEM DE ESTRUTURAS COMO PARÂMETROS

```
#include <stdio.h>

struct ponto
{
    float x;
    float y;
};

void leDados(struct ponto *pp)
{
    printf("Digite as coordenadas do ponto (x y): ");
    scanf("%f %f", &pp->x, &pp->y);
}

void imprime(struct ponto *pp)
{
    printf("O ponto passado foi: (%.2f, %.2f)\n", pp->x, pp->y);
}

//...
```

```
//...
int main() {
    struct ponto p1;

    leDados(&p1);
    imprime(&p1);

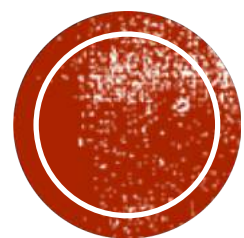
    return 0;
}
```

ALOCAÇÃO DINÂMICA DE ESTRUTURAS

- Por fim, da mesma maneira que vetores, estruturas também podem ser alocadas dinamicamente;
- O procedimento é análogo ao visto anteriormente:

```
...  
struct ponto* p;  
p = (struct ponto*) malloc(sizeof(struct ponto));  
...
```

- No trecho acima, é alocado espaço para armazenamento de um elemento *struct ponto*, e o endereço é armazenado em **p**.



EXERCÍCIOS

EXERCÍCIOS

1. Implemente um programa que faça a avaliação de polinômios de um grau qualquer. Este programa deve ler o grau do polinômio e, em seguida, os coeficientes deste polinômio. Estes coeficientes devem ser alocados em um vetor alocado dinamicamente, de tamanho adequado ao grau do polinômio. Por exemplo, o polinômio $3x^2 + 2x + 12$ tem grau 2 e vetor de coeficientes $v[] = \{12, 2, 3\}$. Por fim, o programa deve ler o valor de x e retornar a avaliação do polinômio.

EXERCÍCIOS

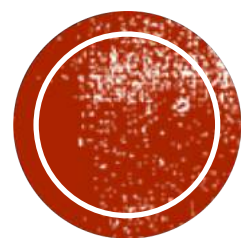
2. Implemente um programa que, utilizando a mesma estrutura do exercício anterior, contenha uma função que retorne o resultado da avaliação da derivada de tal polinômio, aplicada ao valor de x lido. Este valor deve ser impresso na tela ao final.
3. Implemente um programa que leia o número de linhas e colunas de uma matriz, seguido dos valores desta matriz (preenchendo linha a linha). Em seguida, este programa deve verificar se tal matriz é simétrica ou não. Use alocação dinâmica para armazenar a matriz.

EXERCÍCIOS

4. Implemente um programa que contenha uma função que receba uma *string* como parâmetro e retorne uma nova *string*, **alocada dentro da função**, com os caracteres deslocados uma posição para a direita. Por exemplo, a *string* “casa” deve ser convertida na *string* “acas” (note que o último caractere passa para o início da *string*).
5. Crie uma estrutura para armazenar os seguintes dados de um aluno: nome (*string* 100 caracteres), ra (inteiro), nota P1 e nota P2. Em seguida leia os dados de n alunos (n dado pelo usuário) e apresente a média de cada aluno, no formato nome: média. Os dados dos alunos devem ser armazenados em um vetor alocado dinamicamente.

EXERCÍCIOS

- Os seguintes exercícios devem ser entregues via SuSy:
 - 1;
 - 3;
 - 5;
- Veja os enunciados detalhados no site do sistema:
<https://susy.ic.unicamp.br:9999/si100a> (Turma A);
<https://susy.ic.unicamp.br:9999/si100b> (Turma B).



REFERÊNCIAS

REFERÊNCIAS

- CELES, W., CERQUEIRA, R., RANGEL, J. L. Introdução a Estruturas de Dados com Técnicas de Programação em C. Campus, 2004.
- SCHILDT, H. C Completo e Total. 3a Edição, Makron Books, 1997.