

Aula 22

Ponteiros II

Leonardo Garcia Tampelini

Roteiro

- 1 Ponteiros para Registros/Estruturas
- 2 Ponteiros e Alocação Dinâmica
- 3 Ponteiros para Ponteiros e Alocação Dinâmica de Matrizes
- 4 Exercício
- 5 Informações Extras: Organização da Memória do Computador

Ponteiros para Registros

- Ao criarmos uma variável de um tipo **struct**, esta é armazenada na memória como qualquer outra variável, e portanto possui um endereço.
- É possível então criar um ponteiro para uma variável de um tipo **struct**!

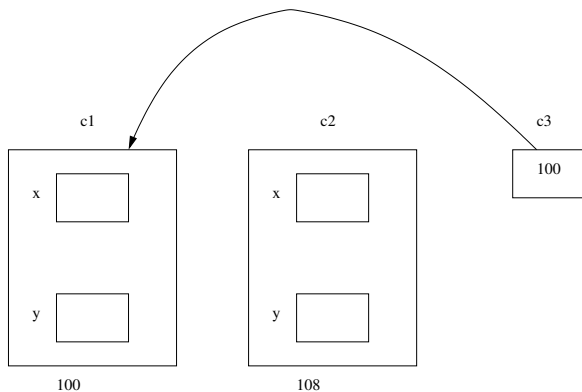
```
#include <stdio.h>

struct Coordenada{
    double x;
    double y;
};

typedef struct Coordenada    Coordenada;

int main(){
    Coordenada c1, c2, *c3;
    c3 = &c1;
    .....
```

Ponteiros para Registros



Ponteiros para Registros

```
#include <stdio.h>
struct Coordenada{
    double x;
    double y;
};
typedef struct Coordenada    Coordenada;

int main(){
    Coordenada c1, c2, *c3;

    c3 = &c1;
    c1.x = -1;
    c1.y = -1.5;

    c2.x = 2.5;
    c2.y = -5;

    *c3 = c2;

    printf("Coordenadas de c1: (%lf,%lf)\n",c1.x, c1.y);
}
```

O que será impresso??

Ponteiros para Registros

- Para acessarmos os campos de uma variável **struct** via um ponteiro, podemos utilizar o operador ***** juntamente com o operador **.** como de costume:

```
Coordenada c1, *c3;  
c3 = &c1;  
(*c3).x = 1.5;  
(*c3).y = 1.5;
```

- Em C também podemos usar o operador **->**, que também é usado para acessar campos de uma estrutura via um ponteiro.

```
Coordenada c1, *c3;  
c3 = &c1;  
c3->x = 1.5;  
c3->y = 1.5;
```

- Resumindo: Para acessar campos de estruturas via ponteiros use um dos dois:
 - ▶ `ponteiroEstrutura->campo`
 - ▶ `(*ponteiroEstrutura).campo`

Ponteiros para Registros

```
int main(){
    Coordenada c1, c2, *c3, *c4;
    c3 = &c1;
    c4 = &c2;

    c1.x = -1;
    c1.y = -1.5;

    c2.x = 2.5;
    c2.y = -5;

    (*c3).x = 1.5;
    (*c3).y = 1.5;

    c4->x = -1;
    c4->y = -1;

    printf("Coordenadas de c1: (%lf,%lf)\n",c1.x, c1.y);
    printf("Coordenadas de c2: (%lf,%lf)\n",c2.x, c2.y);
}
```

O que será impresso??

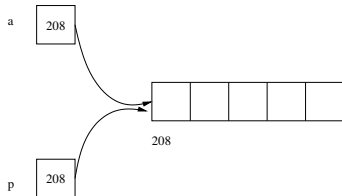
Ponteiros e Alocação Dinâmica

- Lembre-se que uma variável vetor possui um endereço, que podemos atribuí-la para uma variável ponteiro:

```
int a[] = {1, 2, 3, 4, 5};  
int *p;  
p = a;
```

- E podemos então usar **p** como se fosse um vetor:

```
for(i = 0; i<5; i++)  
    p[i] = i*i;
```



Ponteiros e Alocação Dinâmica

- Podemos alocar dinamicamente uma quantidade de memória contígua e associá-la com um ponteiro.
- Desta forma podemos criar programas sem saber a priori o número de dados a ser armazenado.

- ▶ Em aulas anteriores, ao trabalhar com matrizes por exemplo, assumíamos que estas tinham dimensões máximas.

```
#define MAX 100
```

```
.
```

```
.
```

```
.
```

```
int m[MAX][MAX];
```

- ▶ E se o usuário precisar trabalhar com matrizes maiores? Mudar o valor de MAX e recompilar o programa?

Ponteiros e Alocação Dinâmica

Na biblioteca **stdlib.h** existem duas funções para se fazer alocação dinâmica de memória.

- **malloc** : Nesta função é passado um único argumento, o número de bytes que deve ser alocado.

- ▶ Exemplo: alocar 100 inteiros:

```
int *p;  
p = malloc(100*sizeof(int));
```

- **calloc** : Nesta função são passados como parâmetro o número de blocos de memória para ser alocado e o tamanho em bytes de cada bloco. O **calloc** zera todos os bits da memória alocada, enquanto o **malloc** não.

- ▶ Exemplo: alocar 100 inteiros:

```
int *p;  
p = calloc(100, sizeof(int));
```

Ponteiros e Alocação Dinâmica

Juntamente com estas funções, está definida a função **free** na biblioteca **stdlib.h**.

- **free** : Esta função recebe como parâmetro um ponteiro, e libera a memória previamente alocada e apontada pelo ponteiro.

▶ Exemplo:

```
int *p;  
p = calloc(100, sizeof(int));  
....  
free(p);
```

- **Regra para uso correto de alocação dinâmica:** Toda memória alocada durante a execução de um programa e que não for mais utilizada deve ser desalocada (com o **free**)!

Exemplo: Produto escalar de 2 vetores

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
    double *v1, *v2, prodEsc;           int n, i;

    printf("Digite dimensão dos vetores:");
    scanf("%d", &n);
    v1 = malloc(n*sizeof(double));
    v2 = malloc(n*sizeof(double));

    printf("Digite dados de v1: ");
    for(i=0; i<n; i++)
        scanf("%lf", &v1[i]);
    printf("Digite dados de v2: ");
    for(i=0; i<n; i++)
        scanf("%lf", &v2[i]);

    prodEsc=0;
    for(i=0; i<n; i++)
        prodEsc = prodEsc + (v1[i]*v2[i]);

    printf("Resposta: %.2lf\n", prodEsc);
    free(v1);
    free(v2);
}
```

Ponteiros e Alocação Dinâmica

- Você pode fazer ponteiros distintos apontarem para uma mesma região de memória.
 - ▶ Tome cuidado para não utilizar um ponteiro se a região de memória apontada foi desalocada!

```
double *v1, *v2;  
  
v1 = malloc(100 * sizeof(double));  
v2 = v1;  
free(v1);  
  
for(i=0; i<n; i++)  
    v2[i] = i*i;
```

O código acima está errado e pode causar erros durante a execução, já que **v2** está acessando posições de memória que foram desalocadas!

Ponteiros e Alocação Dinâmica

O programa abaixo imprime resultados diferentes dependendo se comentamos ou não o comando **free(v1)**. Por que?

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

int main(){
    double *v1, *v2, *v3;
    int i;

    v1 = malloc(100 * sizeof(double));
    v2 = v1;

    for(i=0; i<100; i++)
        v2[i] = i;
    free(v1);

    v3 = calloc(100,sizeof(double));
    for(i=0; i<100; i++)
        printf("%.2lf\n", v2[i]);
    free(v3);
}
```

Alocação Dinâmica de Matrizes

- Em aplicações científicas e de engenharias, é muito comum a realização de diversas operações sobre matrizes.
- Em situações reais o ideal é alocar memória suficiente para conter os dados a serem tratados. Não usar nem mais e nem menos!
- Como alocar vetores-multidimensionais dinamicamente?

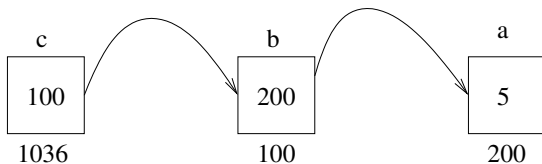
Ponteiros para ponteiros

- Uma variável ponteiro está alocada na memória do computador como qualquer outra variável.
- Portanto podemos criar um ponteiro que contém o endereço de memória de um outro ponteiro.
- Para criar um ponteiro para ponteiro: **tipo **nomePonteiro;**

```
▶ int main(){  
    int a=5, *b, **c;  
    b = &a;  
    c = &b;  
    printf("%d\n", a);  
    printf("%d\n", *b);  
    printf("%d\n", *(*c));  
}
```


Ponteiros para ponteiros

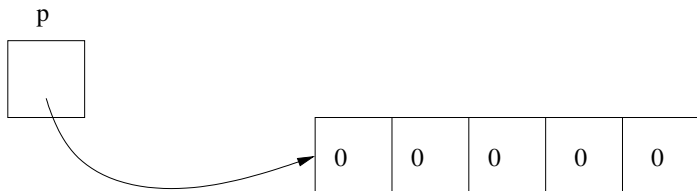
O programa imprime 5 três vezes, mostrando as três formas de acesso à variável **a**: **a**, ***b**, ****c**.



Ponteiros para ponteiros

- Pela nossa discussão anterior sobre ponteiros, sabemos que um ponteiro pode ser usado para referenciar um vetor alocado dinamicamente.

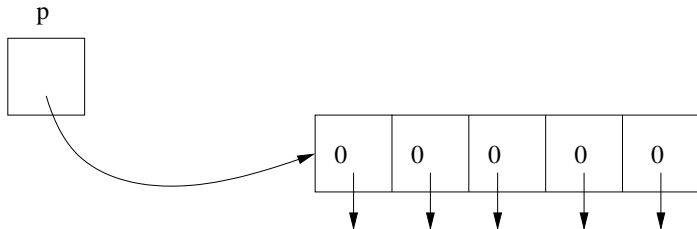
```
▶ int *p;  
  p = calloc(5, sizeof(int));
```



Ponteiros para ponteiros

- A mesma coisa acontece com um ponteiro para ponteiro, só que neste caso o vetor alocado é de ponteiros.

```
► int **p;  
  p = calloc(5, sizeof(int *));
```

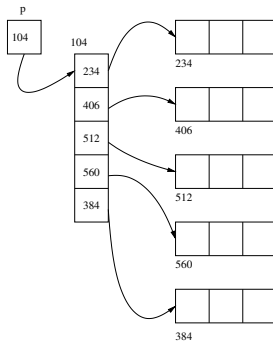


- Note que cada posição do vetor acima é do tipo **int ***, ou seja, um ponteiro para inteiro!

Ponteiros para ponteiros

- Como cada posição do vetor é um ponteiro para inteiro, podemos associar cada posição dinamicamente com um vetor de inteiros!

```
▶ int **p;  
  int i;  
  p = calloc(5, sizeof(int *));  
  
  for(i=0; i<5; i++){  
    p[i] = calloc(3, sizeof(int));  
  }
```



Alocação Dinâmica de Matrizes

Esta é a forma de se criar matrizes dinamicamente:

- Crie um ponteiro para ponteiro.
- Associe um vetor de ponteiros dinamicamente com este ponteiro de ponteiro. O tamanho deste vetor é o número de linhas da matriz.
- Cada posição do vetor será associada com um outro vetor do tipo a ser armazenado. Cada um destes vetores é uma linha da matriz (portanto possui tamanho igual ao número de colunas).

OBS: No final você deve desalocar toda a memória alocada!!

Alocação Dinâmica de Matrizes

```
int main(){
    int **p, i, j;

    p = calloc(5, sizeof(int *));
    for(i=0; i<5; i++){
        p[i] = calloc(3, sizeof(int));
    } //Alocou matriz 5x3

    printf("Digite os valores da matriz\n");
    for(i = 0; i<5; i++)
        for(j=0; j<3; j++)
            scanf("%d", &p[i][j]);

    printf("Matriz lida\n");
    for(i = 0; i<5; i++){
        for(j=0; j<3; j++){
            printf("%d, ", p[i][j]);
        }
        printf("\n");
    }
    //desalocando memória usada
    for(i=0; i<5; i++){
        free(p[i]);
    }
    free(p);
}
```

Alocação Dinâmica de Matrizes

Outro exemplo:

```
int main(){
    int **mat;          int i, j, n, m;

    printf("Numero de linhas:");
    scanf("%d", &n);
    printf("Numero de colunas:");
    scanf("%d", &m);

    mat = malloc(n * sizeof(int *));
    for(i=0; i<n; i++){
        mat[i] = malloc(m * sizeof(int));
    }

    for(i=0; i<n; i++){
        for(j=0; j<m; j++){
            mat[i][j] = i*j;
        }
    }
    .
    .
    .
}
```

Alocação Dinâmica de Matrizes

Outro exemplo:

```
.  
.br/>.br/>for(i=0; i<n; i++){  
    for(j=0; j<m; j++){  
        mat[i][j] = i*j;  
    }  
}  
  
for(i=0; i<n; i++){  
    for(j=0; j<m; j++){  
        printf("%d, ", mat[i][j]);  
    }  
    printf("\n");  
}  
  
for(i=0; i<n; i++){  
    free(mat[i]);  
}  
free(mat);  
}
```


Alocação Dinâmica de Matrizes

Mas a forma mais eficiente de criar matrizes é:

- Para uma matriz de dimensões $n \times m$, crie um vetor unidimensional dinamicamente deste tamanho.
- Use linearização de índices para trabalhar com o vetor como se fosse uma matriz.
- Desta forma tem-se um melhor aproveitamento da cache pois a matriz inteira está sequencialmente em memória.

No final você deve desalocar toda a memória alocada!!

Exercício

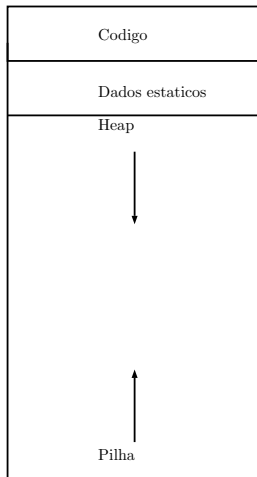
Crie um programa que multiplica duas matrizes quadradas do tipo **double** lidas do teclado. Seu programa deve ler a dimensão n da matriz, em seguida alocar dinamicamente duas matrizes $n \times n$. Depois ler os dados das duas matrizes e imprimir a matriz resultante da multiplicação destas.

Informações Extras: Organização da Memória do Computador

A memória do computador na execução de um programa é organizada em quatro segmentos:

- **Código executável:** Contém o código binário do programa.
- **Dados estáticos:** Contém variáveis globais e estáticas que existem durante toda a execução do programa.
- **Pilha:** Contém as variáveis locais que são criadas na execução de uma função e depois são removidas da pilha ao término da função.
- **Heap:** Contém as variáveis criadas por alocação dinâmica.

Informações Extras: Organização da Memória do Computador



Informações Extras: Organização da Memória do Computador

- Podemos declarar vetores de tamanho não fixo de forma simples declarando este com o tamanho correspondente ao valor de uma variável.
- No programa abaixo, ao invés de declararmos o vetor **v** dinamicamente, declaramos este com o valor da variável **n** que foi lida do teclado.

```
int main(){
    long n, i;

    scanf("%ld", &n);
    double v[n]; //Vetor alocado com tamanho n não pré-estabelecido

    for(i=0; i<n; i++){
        v[i] = i;
    }
    for(i=0; i<n; i++){
        printf("%.2lf\n", v[i]);
    }
}
```

- Execute o programa digitando 1000000 e depois 2000000.

Informações Extras: Organização da Memória do Computador

- O programa anterior será encerrado (*segmentation fault*) se for usado um valor grande o suficiente para n .
- Isto se deve ao fato de que o SO limita o que pode ser alocado na pilha na execução de uma função.
- Este limite não existe para o Heap (com exceção do limite de memória do computador).

Informações Extras: Organização da Memória do Computador

Utilizando alocação dinâmica não temos este problema:

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    long n=2000000, i;
    double *v = malloc(n*sizeof(double));

    for(i=0; i<n; i++){
        v[i] = i;
    }
    for(i=0; i<n; i++){
        printf("%.2lf\n", v[i]);
    }
    free(v);
}
```