

Estruturas, Ponteiros e Ponteiros para Estruturas

Estruturas

- Agrupam informações de diferentes tipos
- Estrutura em C:

```
struct nome_da_estrutura {  
    tipo nome_var1;  
    tipo nome_var2;  
    ...  
};
```

Estruturas (cont.)

➤ Exemplo:

```
struct cliente {  
    char nome[30];  
    char rua[50];  
    int idade;  
};
```

```
struct cliente c1;  
struct cliente c2;
```

Estruturas

```
typedef struct cliente {  
    char nome[30];  
    char rua[50];  
    int idade;  
} tipo_cliente ;
```

```
tipo_cliente c1;  
tipo_cliente c2;
```

Estruturas (cont.)

➤ **Acessando os Campos de uma Estrutura**

```
tipo_cliente c1;
```

```
c1.idade = 10 ;  
gets(c1.nome);
```

Estruturas (cont.)

➤ Passagem de parâmetros por valor

```
void func ( tipo_cliente c){  
    c.idade=10;  
}
```

```
main ( ) {  
    tipo_cliente c1;  
    func(c1);  
}
```

Estruturas (cont.)

Tipo de estrutura

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

typedef struct {
    int a1;
    int a2;
} tipo_vetor;

int
calc_prod_escalar(tipo_vetor
v1, tipo_vetor v2){
    return(v1.a1*v2.a1+v1.a2*v2.
a2);
}
```

```
int main()
{
    tipo_vetor vetor1, vetor2;
    printf("Entre com o vetor 1: ");
    scanf("%d %d",&vetor1.a1,&vetor1.a2);
    printf("Entre com o vetor 2: ");
    scanf("%d %d",&vetor2.a1,&vetor2.a2);

    printf("O produto escalar dos dois vetores
eh: %d\n", calc_prod_escalar(vetor1,vetor2));

    printf("O produto vetorial dos dois vetores
eh: %d\n", calc_prod_vetorial(vetor1,vetor2));

    return 0;
}
```

Estruturas (cont.)

Tipo de estrutura

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

```
typedef struct {
    int a1;
    int a2;
} tipo_vetor;
```

```
int
calc_prod_escalar(tipo_vetor
v1, tipo_vetor v2){
    return(v1.a1*v2.a1+v1.a2*v2.
a2);
}
```

```
int main()
{
    tipo_vetor vetor1, vetor2;
    printf("Entre com o vetor 1: ");
    scanf("%d %d",&vetor1.a1,&vetor1.a2);
    printf("Entre com o vetor 2: ");
    scanf("%d %d",&vetor2.a1,&vetor2.a2);
    printf("O produto escalar dos dois vetores
eh: %d\n", calc_prod_escalar(vetor1,vetor2));
    return 0;
}
```


Exemplo de definição de estruturas

```
struct tipo_vetor {  
    int a1;  
    int a2;  
};  
struct tipo_vetor vetor1, vetor2;
```

```
typedef struct tipo_vetor {  
    int a1;  
    int a2;  
};  
tipo_vetor vetor1, vetor2;
```

```
struct {  
    int a1;  
    int a2;  
} vetor1, vetor2;
```

estrutura anônima, que não pode ser referenciado em outras partes do programa, ou seja, não é possível definir outras variáveis usando esta estrutura.

Estruturas (cont.)

- É possível criar um *tipo de estrutura* em que um ou mais de seus membros também sejam estruturas.
- ✓ Os tipos de tais estruturas devem ter sido previamente declarados no programa.

Estruturas (cont.)

➤ Exemplo :

```
struct tipo_endereco  
{  
    char rua [35];  
    int numero;  
    char bairro [20];  
    char cidade [30];  
    char estado [3];  
    char cep[10];  
};
```

```
struct Ficha {  
    char Nome [45];  
    char Fone[15];  
    struct tipo_endereco ender;  
    char cargo[30];  
    float salario;  
    char sexo;  
};
```

```
void main () {  
    struct Ficha pessoa;  
    printf("\Digite o CEP da pessoa: ");  
    gets(pessoa.ender.cep);  
    printf("\Digite o sexo da pessoa",);  
    pessoa.sexo = getche();  
}
```

Ponteiros

- Ponteiro é uma variável que armazena um endereço de memória.
- Declaração de ponteiros:

`tipo *nome;`

→ nome da variável

operador especial para denotar que a variável armazena um endereço de memória

qualquer tipo válido em C

- **Exemplos:**
- `float *f;` // *f é um ponteiro para variáveis do tipo float*
- `int *p;` // *p é um ponteiro para variáveis do tipo inteiro*

Ponteiros (cont.)

& operador unário que devolve o endereço de memória de seu operando

- `int count, q, *m;`
- `m = &count;`

Lê-se: “m recebe o endereço de count”

Se a variável **count** está armazenada na posição de memória **1000**, o valor de **m** será **1000**

Ponteiros (cont.)

* operador unário que devolve o valor da variável localizada no endereço que o segue

➤ `int count, q, *m;`

➤ `q=*m;`

Lê-se: “q recebe o valor que está no endereço m”

Se a variável `m` aponta para um endereço que contém o valor 200, o valor de `q` será 200.

Ponteiros (cont.)

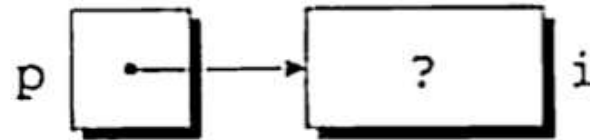
**** ATRIBUIÇÃO COM PONTEIROS***

Exemplo:

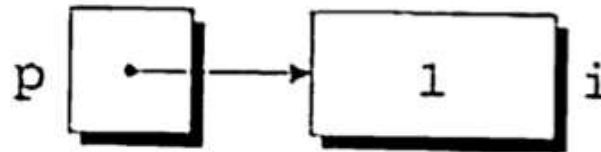
```
int x, *p1, *p2;  
p1 = &x; //p1 aponta para x  
p2 = p1; //p2 recebe p1 e passa a apontar para x  
printf("%p", p2); //escreve o endereço de x
```

Ponteiros (cont.)

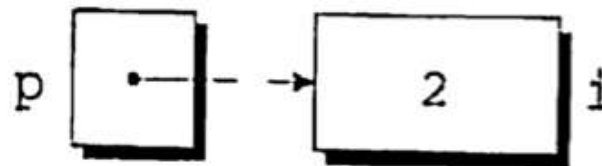
```
p = &i;
```



```
i = 1;
```



```
printf("%d\n", i); /* imprime 1 */  
printf("%d\n", *p); /* imprime 1 */  
*p = 2;
```



```
printf("%d\n", i); /* imprime 2 */  
printf("%d\n", *p); /* imprime 2 */
```


Ponteiros (cont.)

INICIALIZAÇÃO DE PONTEIROS

Após ser declarado, e antes de receber um valor, o valor de um ponteiro é desconhecido.

- Um ponteiro que não aponta para um local de memória válido recebe o valor nulo (NULL)

- Exemplo: `p = NULL;`

Exemplo

```
int x, *p;  
p=null;  
x = 10;  
*p = x;
```

ERRO

Ponteiros (cont.)

INICIALIZAÇÃO DE PONTEIROS

Após ser declarado, e antes de receber um valor, o valor de um ponteiro é desconhecido.

- Um ponteiro que não aponta para um local de memória válido recebe o valor nulo (NULL)

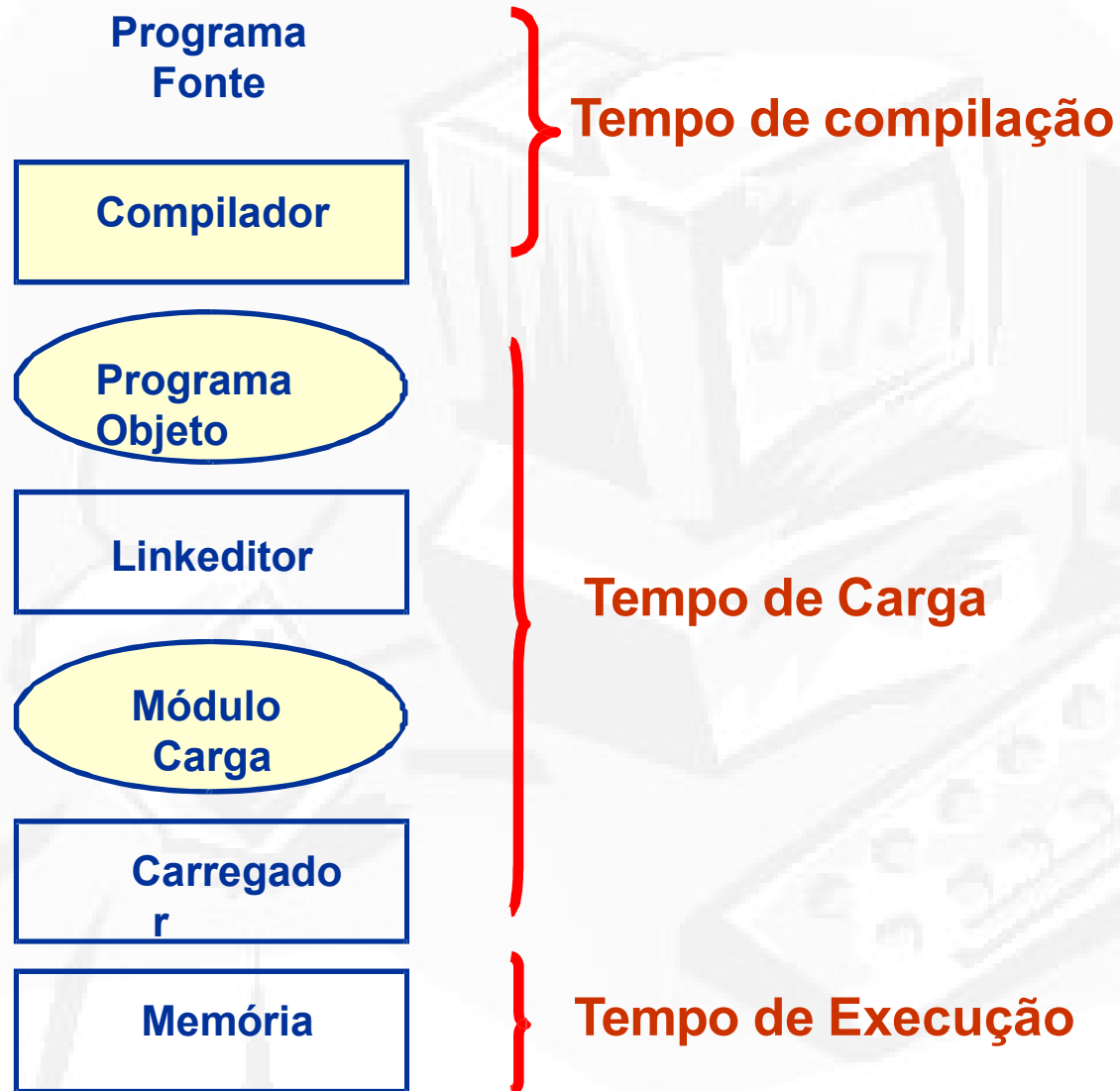
- Exemplo: `p = NULL;`

Exemplo

```
int x, *p;  
p=null;  
x = 10;  
p = &x;
```

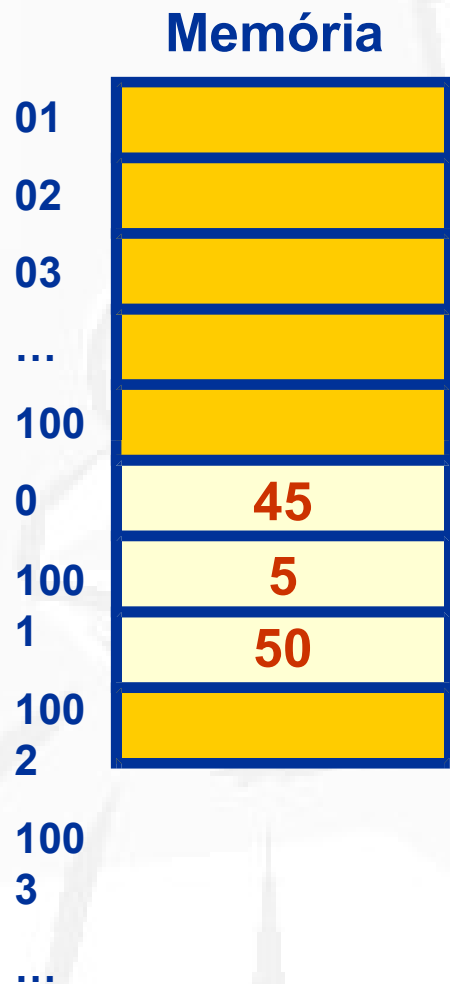
Correto

Alocação de memória



Alocação de memória

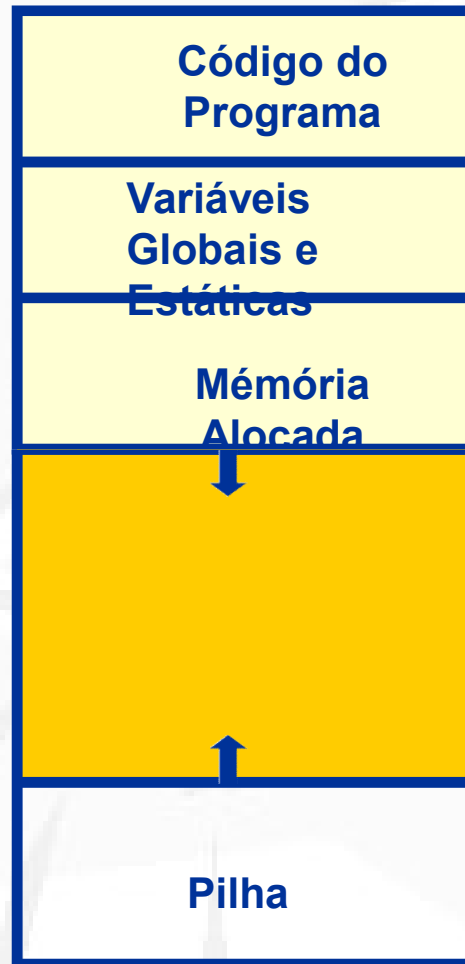
Memória Principal: visão simplificada



SO carrega o programa a partir de uma determinada posição da memória;
Nesse espaço, variáveis declaradas são sinônimos de endereço;
Comandos fazem referências a endereços, portanto, operações referenciam esses endereços.

$$X = K + N$$
$$[1003] = [1001] + [1002]$$
$$50 = 45 + 5$$

Alocação de memória

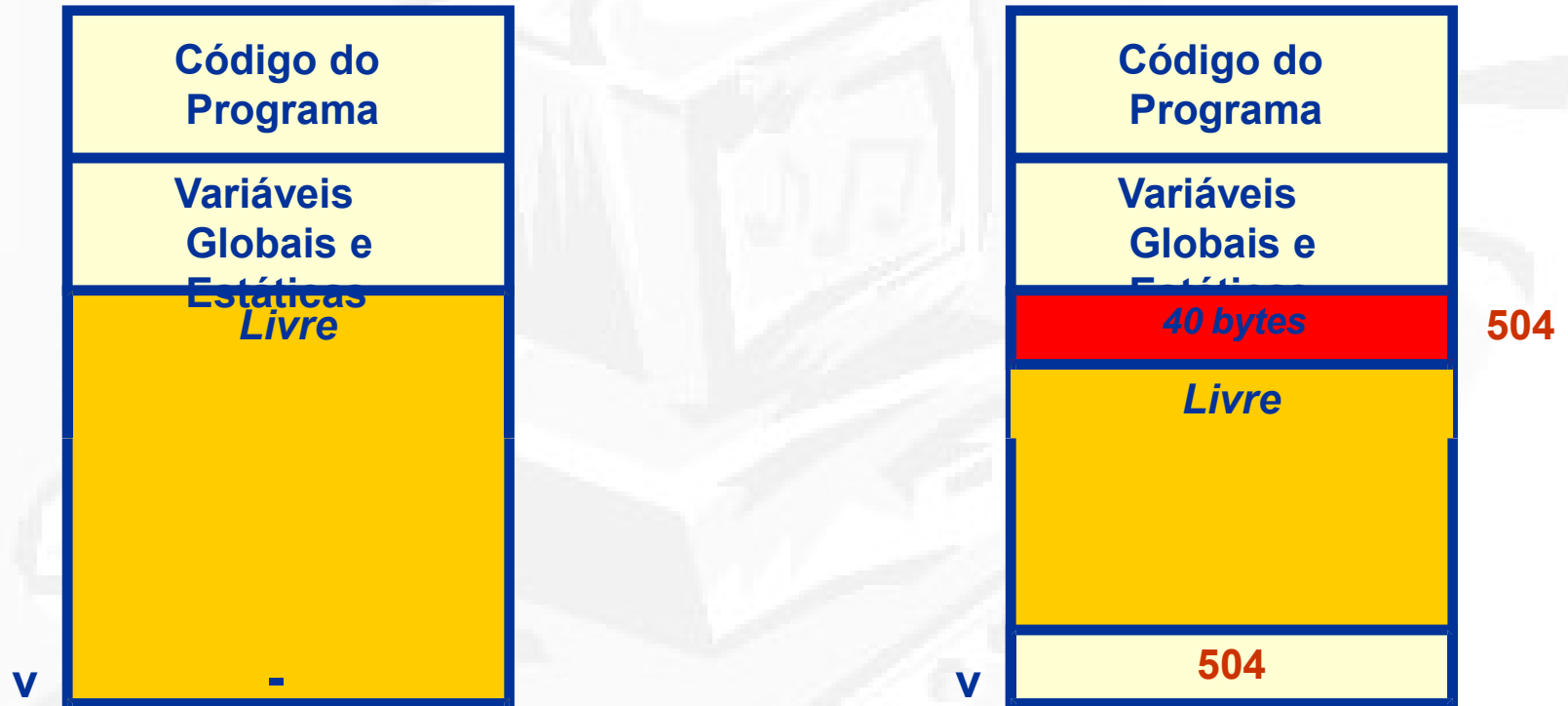


Uso de Memória

Memória Livre

Alocação de memória

Uso de Memória



Alocação estática de memória

```
int main(){  
    int vet[100]; ← declaração  
    ...  
    scanf("%d",&vet[8]); ← referência  
    ...  
}
```



vet

**Permanece alocado
durante toda a
execução do
programa**

**É preciso prever todo o espaço
necessário antes da execução do
programa** !

Alocação dinâmica de memória

- espaço de memória alocado e liberado **durante a execução do programa**
- **erro de execução** caso não exista espaço suficiente
- **tamanho e tipo de área de memória alocada** informados no momento da solicitação
- Cuidar para não perder o endereço das variáveis alocadas dinamicamente

Alocação dinâmica

➤ Em C há 3 funções declaradas na biblioteca `<stdlib.h>` que podem ser usadas para alocar memória dinamicamente (em tempo de execução):

- ✓ `malloc`: aloca um bloco de memória, mas não o inicializa (mais utilizada);
`void *malloc (size_t size);`
- ✓ `calloc`: aloca um bloco de memória e o inicializa com zero (menos eficiente que `malloc`);
`void *calloc (size_t n_element, size_t size);`
- ✓ `realloc`: redimensiona um bloco de memória previamente alocado.
`void *realloc (void *ptr, size_t size);`

Alocação dinâmica (cont.)

- A função **malloc** aloca um determinado número de bytes na memória, retornando um ponteiro para o primeiro byte alocado, ou **NULL** caso não tenha conseguido alocar.
- A função **free**, por outro lado, libera o espaço alocado.

```
p = malloc(1000);  
if(p==NULL) {  
    // tratamento para falha de alocação  
}  
...  
free(p);
```

Exemplo

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

int main()
{
    int *p;
    p = (int *)malloc(sizeof(int));
    *p=10;
    printf("valor P = %d \n",*p);
    printf("endereco P = %d \n",p);
    free(p);
    return 0;
}
```

Exemplo

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

int main(void)
{
    char *c; /* ponteiro para o espaço alocado */

    c = (char *)malloc(1); /* aloco um único byte
                           na memória */
    if (c == NULL) { /* testa se conseguiu alocar.
                     Equivalente a "if (!c)" */
        printf("Não conseguiu alocar a memória\n");
        exit(1);
    }

    *c = 'd'; /* carrego um valor na região
               de memória alocada */
    printf("%c\n", *c); /* escrevo este valor */
    free(c); /* libero a memória alocada */
    return 0;
}
```

Estruturas e Ponteiros

➤ Passagem de parâmetros por referência

```
void func ( tipo_cliente *c){  
  *c.idade=10 ERRO  
}
```

```
main ( ) {  
  tipo_cliente c1;  
  func(&c1);  
}
```

Estruturas e Ponteiros

➤ Passagem de parâmetros por referência

```
void func ( tipo_cliente *c){  
  (*c).idade=10 ou c->nome ERRO  
}
```

```
main ( ) {  
  tipo_cliente c1;  
  func(&c1);  
}
```

Ponteiro para estruturas

- Podemos também declarar variáveis do tipo ponteiro para estruturas:

```
struct ponto p {  
    double x;  
    double y;  
};
```

```
struct ponto *pp;
```

ou

```
typedef  
ponto *Ponto;
```

```
struct
```

```
...
```

```
Ponto pp;
```

Alocação dinâmica de estruturas

- As estruturas podem ser alocadas dinamicamente:

```
struct ponto *p;
```

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

```
p->x = 12.0;
```


Vetores para estruturas

- Suponha que um programa funcione como um servidor e permita até 10 usuários conectados simultaneamente. Poderíamos guardar as informações desses usuários num vetor de 10 estruturas:

```
struct info_usuario {  
    int id;  
    char nome[20];  
    long endereco_ip;  
    time_t hora_conexao;  
};  
  
struct info_usuario usuarios[10];
```

Vetores para estruturas (cont.)

- Suponha que um programa funcione como um servidor e permita até 10 usuários conectados simultaneamente. Poderíamos guardar as informações desses usuários num vetor de 10 estruturas:

```
struct info_usuario {  
    int id;  
    char nome[20];  
    long endereco_ip;  
    time_t hora_conexao;  
};  
  
struct info_usuario usuarios[10];
```

Para obter o horário em que o 2º usuário se conectou:

```
usuarios[1].hora_conexao.
```

Vetores de ponteiros para estruturas

- O uso de vetores de ponteiros é interessante quando temos que tratar um conjunto de elementos complexos.
- Ex: considere que se deseja armazenar uma tabela com dados de alunos. Podemos organizar tais dados em um vetor:

```
struct aluno {  
    char nome[81];  
    int mat;  
    char end[121];  
    char tel[21];  
};  
typedef struct aluno Aluno;  
  
Aluno tab[100];  
.....  
tab[i].mat = 99122321;
```

Vetores de ponteiros para estruturas

- Desvantagem: o tipo `aluno` como definido ocupa pelo menos 227 (81+4+121+21) bytes, o que representa um desperdício significativo de memória, uma vez que provavelmente será armazenado um número de alunos bem inferior ao máximo estimado.
- Alternativa: utilizar vetor de ponteiros.

```
typedef struct aluno *PAluno;  
PAluno tab[100];
```
- Assim, cada elemento do vetor ocupa apenas o espaço necessário para armazenar um ponteiro. Quando precisarmos alocar os dados de um aluno numa determinada posição do vetor, alocamos dinamicamente a estrutura `Aluno` e guardamos seu endereço no vetor de ponteiros.

Exercício 1

- Faça um programa que use uma estrutura do tipo CD e faça a leitura de 10 cds (use um vetor de estrutura do tipo cd). A estrutura CD deve conter os seguintes campos:

Nome da banda

Data do lançamento do cd: dia, mês e ano

Data da contratação da empresa: dia, mês e ano

Valor do cd

Número de membros da banda

Produtora do cd

```
struct Data
{
int dia;
int mês;
int ano;
};
struct Banda{
char nome[10];
struct Data Lancamento;
struct Data Contratacao;
float valor;
int membros;
char produtora[15];
}
main() {
int i;
struct Banda Bandas[10]; //não há o que alocar, pois não foi usado ponteiro
for(i=0;i<10;i++)
{
scanf("%d",&Bandas[i].membros);
scanf("%f",&Bandas[i].valor);
scanf("%s",&Bandas[i].nome);
scanf("%s",&Bandas[i].produtora);
scanf("%d",&Bandas[i].Lançamento.dia);
scanf("%d",&Bandas[i].Lançamento.mes);
scanf("%d",&Bandas[i].Lançamento.ano);
scanf("%d",&Bandas[i].Contratacao.dia);
scanf("%d",&Bandas[i].Contratacao.mes);
scanf("%d",&Bandas[i].Contratacao.ano);
}
}
```

Exercício 2

- Refaça o programa anterior utilizando ponteiro para estrutura e considerando que há dados de n bandas a serem lidos (n é entrada pelo usuário).

```

struct Data
{
int dia;
int mês;
int ano;
};
struct Banda{
char nome[10];
struct Data Lancamento;
struct Data Contratacao;
float valor;
int membros;
char produtora[15];
}
int main()
{
int n, i;
struct Banda *Bandas[100];

```

```

    printf("Digite a quantidade de bandas que sera considerado: ");
    scanf("%d",&n);
// primeiro temos que alocar os espaços para armazenar os dados
// e só depois podemos utilizar os campos dos registros

```

```

for(i=0;i<n;i++)
{
    Bandas[i] = (struct Banda *) malloc(sizeof(struct Banda));
    scanf("%d",&Bandas[i]->membros);
    scanf("%f",&Bandas[i]->valor);
    scanf("%s",&Bandas[i]->nome);
    scanf("%s",&Bandas[i]->produtora);
    scanf("%d",&Bandas[i]->Lancamento.dia);
    scanf("%d",&Bandas[i]->Lancamento.mes);
    scanf("%d",&Bandas[i]->Lancamento.ano);
    scanf("%d",&Bandas[i]->Contratacao.dia);
    scanf("%d",&Bandas[i]->Contratacao.mes);
    scanf("%d",&Bandas[i]->Contratacao.ano);
}
// após manipular os dados é necessário desalocar os espaços alocados
for(i=0;i<n;i++)
    free(Bandas[i]);
return(0);
}

```


Exercício 3

- Considerando a estrutura Aluno definida anteriormente, escreva um programa que contenha uma função para inicializar uma tabela de alunos, uma função que armazena os dados de um novo aluno numa dada posição do vetor, uma que mostre as informações do aluno numa dada posição do vetor (esta função deve prever o caso de tentar mostrar os dados em uma posição sem dados) e o programa principal. Utiliza ponteiro para estrutura para este programa.

```

/*Vetores de ponteiros para estruturas do
    tipo Aluno*/
#include <stdio.h>
#define TAM 10

struct aluno{
    int matricula;
    char nome[10];
}
typedef struct aluno *Paluno;

//Alocação
void aloca(Paluno al, int pos) {
    al[pos] = (Aluno*)malloc(sizeof(Aluno));
}

//alocacao e leitura dos dados de um aluno,
    o aluno i
void leitura(Paluno al, int pos){
    if (al[pos]==NULL){
        printf("\n Matricula: ");
        scanf("%d", &al[pos]->matricula);
        printf("\n Nome: ");
        scanf("%s", al[pos]->nome);
    }
}

```

```

//desaloca espaços alocados
void desaloca(Paluno al, int n){
    for(i=0;i<n;i++)
        free(al[i]);
}

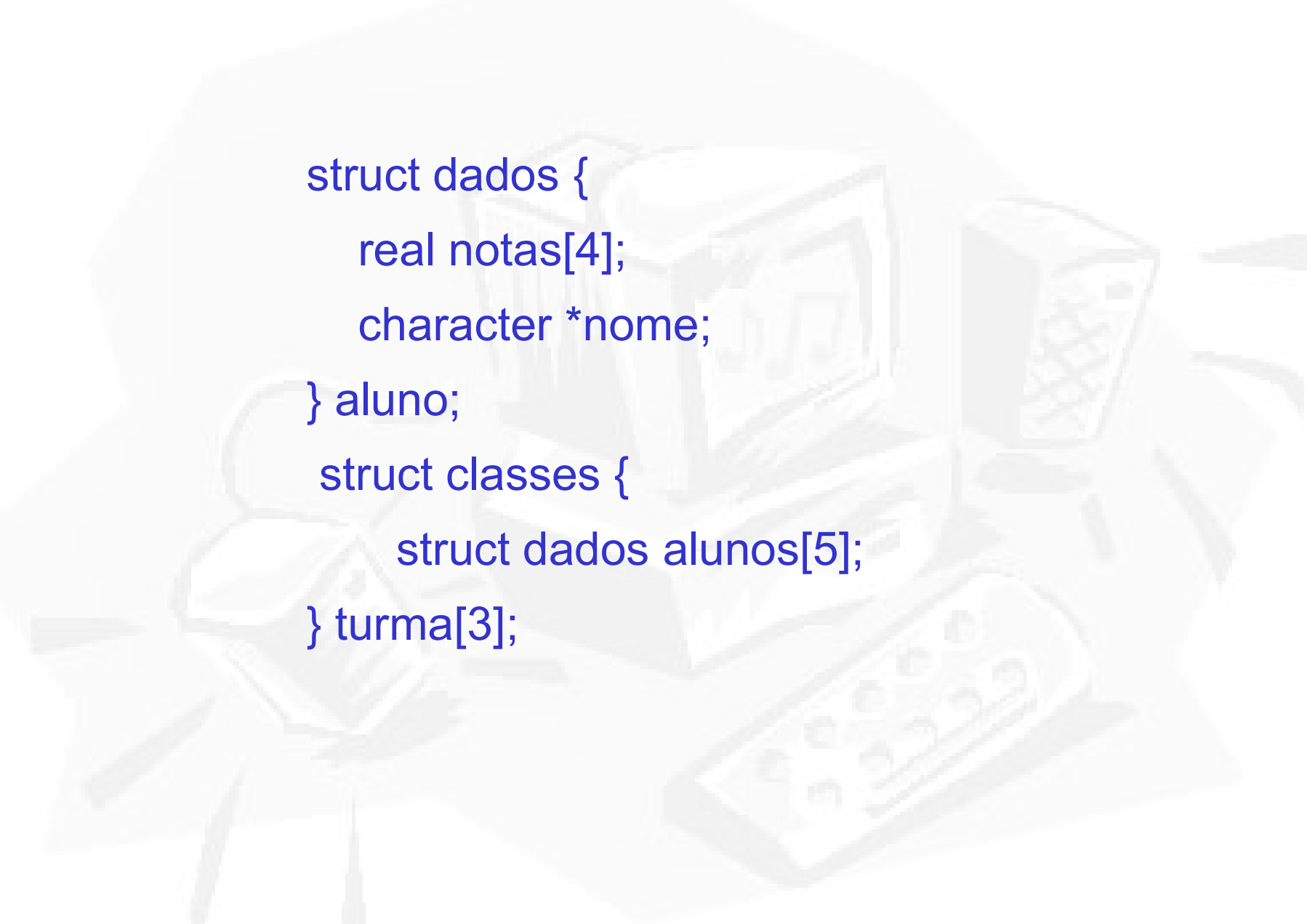
//mostrar as informações do aluno no indice pos
void mostrar(Paluno al, int pos){
    if (al[pos]==NULL)
        printf("\n Aluno inexistente");
    else
        printf("\n Matricula: %d Nome: %s", al[pos]-
>matricula, al[pos]->nome);
}

//funcao main
int main(){
    Aluno *al[TAM];
    int i = 0, j;
    char resp;
    do {
        aloca(al,i);
        leitura(al, i);
        i++;
        printf ("Deseja cadastrar mais aluno? (s/n) ")
        resp = getchar();
        while (resp == 's' || resp == 'S');
        for (j=0;j<i;j++)    mostrar(al, j);
        desaloca(al,i);
    }
    return 0;
}

```

Exercício 4

- Elabore um programa em C para calcular a média das 4 notas bimestrais de cada aluno (usando registros), para um professor que tenha 3 turmas de 5 alunos. Inclua no programa a possibilidade de procurar um registro (dentre os digitados) pelo nome da pessoa, e apresentar seus dados na tela. Inclua também a possibilidade de excluir um registro que possua o campo nome igual ao valor passado pelo usuário.



```
struct dados {  
    real notas[4];  
    character *nome;  
}  
aluno;  
  
struct classes {  
    struct dados alunos[5];  
}  
turma[3];
```

Exercício 5

- Suponha dois vetores, um de registros de estudantes e outro de registros de funcionários. Cada registro de estudante contém campos para o último nome, o primeiro nome e um índice de pontos de graduação. Cada registro de funcionário contém membros para o último nome, primeiro nome e o salário. Ambos os vetores são classificados em ordem alfabética pelo último e pelo primeiro nome. Dois registros com o último e o primeiro nome iguais não aparecem no mesmo vetor. Faça um programa em C para conceder um aumento de 10% a todo funcionário que tenha um registro de estudante cujo índice de pontos de graduação seja maior que 3.0.
- **Obs:** Não é necessário implementar a ordenação. Considere que os dados já são entrados pelo usuário na ordem correta.

Referências

- Slides adaptados de Regina Célia
- Livro:
 - ✓ TENENBAUM, Aaron M et al. Estruturas de dados usando C. São Paulo: Pearson, 2008. 884 p. ISBN 978-85-346-0348-5.