

Estruturas, Ponteiros e Ponteiros para Estruturas

Estruturas

- **Estrutura em C:**

```
struct (nome_da_estrutura) {  
    tipo nome_var1;  
    tipo nome_var2;  
    ...  
} variável_tipo_nomedaestrutura;
```

- **Estruturas** também são conhecidas como **registros** em outras linguagens.
- **Membros** são também denominados **campos**.

Estruturas (cont.)

➤ Exemplo:

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

- ✓ Neste exemplo, foi declarado um tipo ***struct cliente***.
- ✓ A partir deste ponto, poderão ser declaradas variáveis do tipo ***struct cliente***.
- ✓ Primeira variável criada: ***voce***.
- ✓ Se quisermos declarar outras variáveis deste mesmo tipo:
struct cliente eu;

Estruturas (cont.)

➤ Acessando os Campos de uma Estrutura

```
struct cliente eu;
```

```
eu.idade = 10; // campo idade da variável eu recebe o  
// valor 10
```

- ★ O acesso aos campos de uma estrutura é semelhante ao acesso nos vetores. Só que nos vetores a localização dos campos é feita pelo índice e nas estruturas isso é feito com o nome dos campos.

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

```
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;
}
```

Exemplo de definição de estruturas

```
struct tipo_vetor {  
    int a1;  
    int a2;  
};  
  
struct 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();
}
```

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

Exercício 2

- Escreva um trecho de código para fazer a criação de novos tipos de dados conforme solicitado a seguir:
 - ✓ Horário: composto de hora, minutos e segundos.
 - ✓ Data: composto de dia, mês e ano.
 - ✓ Compromisso: composto de uma data, horário e texto que descreve o compromisso.

Ponteiros

- Ponteiro é uma variável que armazena um endereço de memória.
- Devem ser declarados usando o símbolo “`*`”:

```
int *p;
```

✓ `p` é o nome da variável que armazenará um endereço de memória e `int *` informa o compilador que `p` armazenará um endereço de memória em que será guardado um número inteiro.

Ponteiros (*cont.*)

- É possível declarar ponteiro junto com outras variáveis:

```
int i, *p, j, v[10], *q;
```

- Há dois operadores que são usados com ponteiros:
 - ✓ Operador de endereço: &
 - ★ Se x é variável, então $\&x$ é o endereço de memória de x .
 - ✓ Operador indireto: *
 - ★ Se p é um ponteiro, então $*p$ é o conteúdo armazenado no endereço que p guarda.

Ponteiros (*cont.*)

- Declarar uma variável ponteiro reserva espaço na memória para o apontador, mas não faz referência a nenhum objeto:

```
int *p; // não guarda nada ainda
```

- Para usar *p*, primeiro é necessário inicializá-lo ou alocar um espaço de memória para que ele receba o endereço desse espaço alocado:

```
int *p, i;
```

```
...
```

```
p = &i;
```

```
int i, *p = &i;
```

ou



Ponteiros (*cont.*)

- É possível inicializar um ponteiro vazio fazendo:

```
int *p;
```

```
p = NULL; // NULL é definida em stdlib.h
```

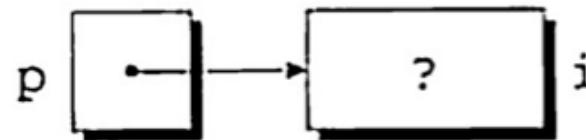
- Uma vez que o ponteiro aponta para um objeto, é possível usar o operador ***** para acessar seu conteúdo:

```
printf("%d\n", *p); // imprime o conteúdo armazenado no  
endereço que p guarda
```

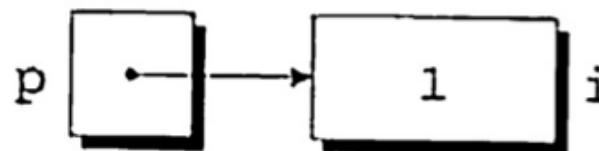
- Se **p** aponta para **i**, ***p** é dito ser um *alias* de **i**.
 - ✓ ***p** tem o mesmo valor que **i**.
 - ✓ Alterar ***p** também altera o valor de **i**.

Ponteiros (*cont.*)

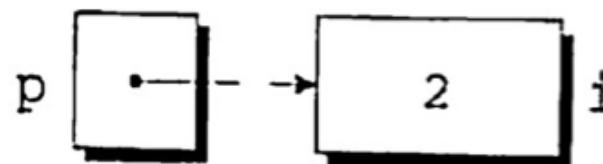
p = &i;



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.)

- Nunca aplique um operador indireto (*) em um apontador não inicializado.

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

✓ Pode causar um comportamento indefinido.

- Nunca atribua um valor que não seja um endereço a uma variável do tipo ponteiro, pois poderá travar o sistema.

```
int *p;  
p = 1; //NUNCA FAÇA ISSO!!
```

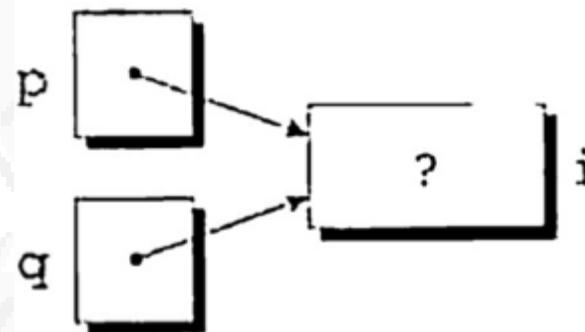
Ponteiros (*cont.*)

- C permite o operador de atribuição para copiar endereços:

```
int i, j, *p, *q;
```

```
p = &i; //endereço de i é copiado para p
```

```
q = p; // endereço de p é copiado para q
```

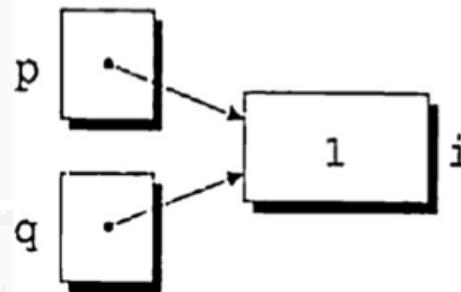


- Tanto *p* quanto *q* agora apontam para *i* (guardam o endereço de memória de *i*).

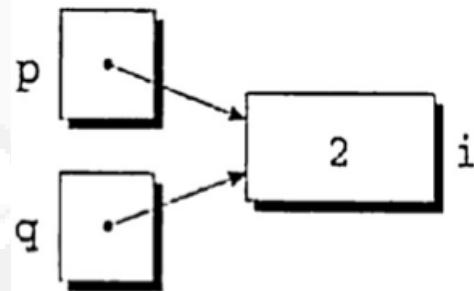
Ponteiros (cont.)

- Agora é possível alterar o valor de i atribuindo novos valores para $*p$ e $*q$:

$*p = 1;$



$*q = 2;$



- Qualquer número de ponteiros pode apontar para o mesmo objeto.

Ponteiros (*cont.*)

➤ Cuidado!!! Não confunda:

$p = q;$

com

$*q = *p;$

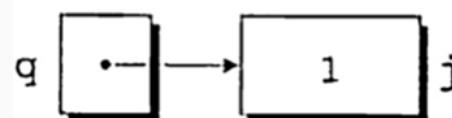
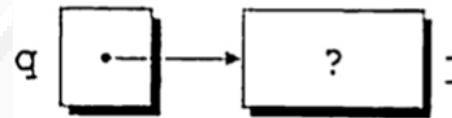
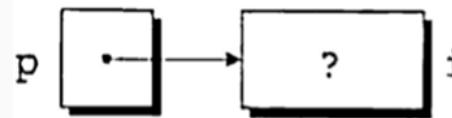
➤ O primeiro é uma atribuição, enquanto o segundo não.

$p = \&i;$

$q = \&j;$

$i = 1;$

$*q = *p;$



➤ $*q = *p$ copia o valor da variável apontada por p (valor de i) dentro do objeto apontado por q (variável j)

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 <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;
}
```

Alocação dinâmica (cont.)

- Declaramos um ponteiro para caractere e usamos **malloc** para alocar um byte na memória (tamanho de um caractere).
- Quando **malloc** aloca a memória, ele não faz ideia do tipo de elemento que será, portanto retorna um ponteiro genérico (**void ***).
- Contudo, C precisa que o ponteiro tenha um tipo, para poder executar operações de aritmética de ponteiros. Por isso **temos que fazer um cast no retorno de malloc** para o tipo de ponteiro. No nosso caso, como queríamos um ponteiro para caractere, forçamos a saída de malloc a ser **char ***.

Alocação dinâmica (cont.)

- No caso de `c` ser `NULL`, é impressa uma mensagem de erro e encerramos o programa. Se `c` não for `NULL`, então ele contém o endereço na memória que cabe um caractere. Agora é só agir como faríamos com um ponteiro que apontou para uma variável char, guardando um valor na região de memória apontada.
- Antes do encerramento do programa, é necessário liberar a memória alocada, usando a função `free`.

Ponteiro para estruturas

- Podemos também declarar variáveis do tipo ponteiro para estruturas:

```
struct ponto p {  
    double x; double y;  
}  
  
struct ponto *pp;          ou      typedef struct ponto *Ponto;  
  
...  
  
Ponto pp;
```

- Para acessar os campos de um ponteiro para estrutura utilizamos o operador **->**:

(*pp).x = 12.0; ou pp->x = 12.0;

Ponteiro para estruturas (cont.)

- Passagem de parâmetros de estruturas para funções funciona de forma análoga à passagem de variáveis simples.
- Se a estrutura for passada por valor como parâmetro da função, não há como alterar os valores dos elementos.
- É mais eficiente sempre passar o ponteiro da estrutura como parâmetro (ocupa, em geral, 4 bytes) que a estrutura inteira, mesmo quando não for alterar o valor dos elementos.

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 (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].horaconexao.

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 3

- Refaça o programa anterior da banda de músicas (exercício 1) utilizando ponteiro para estrutura e considerando que há dados de n bandas a serem lidas (n é entrado pelo usuário).

Exercício 4

- 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.

Exercício 5

- 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 (entre 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.

Exercício 6

- 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 sobre ponteiros: Prof. Tiago de Oliveira
- Livro:
 - ✓ TENENBAUM, Aaron M et al. Estruturas de dados usando C. São Paulo: Pearson, 2008. 884 p. ISBN 978-85-346-0348-5.