LPG0002 – Linguagem de Programação

Tipos Estruturados
(parte 2)

Profª Luciana Rita Guedes
Departamento de Ciência da Computação
UDESC / Joinville

Material elaborado por: Prof. Rui Jorge Tramontin Junior

Introdução

- **Estruturas** também podem ser passadas como parâmetros para funções;

Introdução

- **Estruturas** também podem ser passadas como parâmetros para funções;
  - *Por valor*;
  - *Por referência (ponteiro para a estrutura)*;

Introdução

- **Estruturas** também podem ser passadas como parâmetros para funções;
  - *Por valor*;
  - *Por referência (ponteiro para a estrutura)*;

- Uma função também pode ter uma estrutura como tipo de retorno;

Introdução

- **Estruturas** também podem ser passadas como parâmetros para funções;
  - *Por valor*;
  - *Por referência (ponteiro para a estrutura)*;

- Uma função também pode ter uma estrutura como tipo de retorno;

- O uso de funções com estruturas ajuda a tornar o código mais abstrato;

Exemplo 1: mostrar produto

```
struct Produto {
    int codigo;
    char descricao[20];
    float preco;
};
```

Exemplo 1: mostrar produto

```
struct Produto {
    int codigo;
    char descricao[20];
    float preco;
};


void mostra_produto ( struct Produto x ){



}
```

Exemplo 1: mostrar produto

```
struct Produto {
    int codigo;
    char descricao[20];
    float preco;
};


void mostra_produto ( struct Produto x ){
    printf("Código: %d\n", x.codigo);
    printf("Descrição: %s\n", x.descricao);
    printf("Preço: R$%.2f\n\n", x.preco);
}
```

Exemplo 1: mostrar produto

```
int main(){
    struct Produto k;




}
```

Exemplo 1: mostrar produto

```
int main(){
    struct Produto k;

    k.codigo = 123;
    strcpy(k.descricao, "Caderno");
    k.preco = 10.0;


}
```

Exemplo 1: mostrar produto

```
int main(){
    struct Produto k;

    k.codigo = 123;
    strcpy(k.descricao, "Caderno");
    k.preco = 10.0;

    mostra_produto( k );
}
```

Considerações

- Assim como variáveis de tipo simples, estruturas podem ser passadas por valor;

Considerações

- Assim como variáveis de tipo simples, estruturas podem ser passadas por valor;

- O parâmetro ***x*** da função também é uma estrutura, e recebe uma cópia da variável ***k*** da *main()*;

Considerações

- Assim como variáveis de tipo simples, estruturas podem ser passadas por valor;

- O parâmetro ***x*** da função também é uma estrutura, e recebe uma cópia da variável ***k*** da *main()*;

- Portanto, qualquer mudança dentro da função não afetaria a variável na *main()*.

Ponteiro para Estrutura

- Para modificar uma estrutura dentro da função é preciso fazer *passagem por referência*

Ponteiro para Estrutura

- Para modificar uma estrutura dentro da função é preciso fazer *passagem por referência* → **ponteiro para a estrutura**;

Ponteiro para Estrutura

- Para modificar uma estrutura dentro da função é preciso fazer *passagem por referência* → **ponteiro para a estrutura**;

- Um ponteiro para estrutura é declarado tal como um ponteiro para tipo simples:

Ponteiro para Estrutura

- Para modificar uma estrutura dentro da função é preciso fazer *passagem por referência* → **ponteiro para a estrutura**;

- Um ponteiro para estrutura é declarado tal como um ponteiro para tipo simples:

```
struct Produto *p;
```

Ponteiro para Estrutura

- O acesso aos campos da estrutura pode ser feito de duas formas;

Ponteiro para Estrutura

- O acesso aos campos da estrutura pode ser feito de duas formas;

- Operador *seta* (**->**):

Ponteiro para Estrutura

- O acesso aos campos da estrutura pode ser feito de duas formas;

- Operador *seta* (->):

```
p->codigo = 123;
```

Ponteiro para Estrutura

- O acesso aos campos da estrutura pode ser feito de duas formas;
- Operador *seta* (->):

```
p->codigo = 123;
```

- Operador de indireção (*), para resolver o ponteiro, e o operador de acesso (.):

Ponteiro para Estrutura

- O acesso aos campos da estrutura pode ser feito de duas formas;
- Operador *seta* (`->`):

```
p->codigo = 123;
```

- Operador de indireção (`*`), para resolver o ponteiro, e o operador de acesso (`.`):

```
(*p).codigo = 123;
```

Exemplo 2: passagem por referência

```
struct Produto {
    int codigo;
    char descricao[20];
    float preco;
};
```

Exemplo 2: passagem por referência

```
struct Produto {
    int codigo;
    char descricao[20];
    float preco;
};

void le_produto( struct Produto *p ){



}
```

Exemplo 2: passagem por referência

```
struct Produto {
    int codigo;
    char descricao[20];
    float preco;
};

void le_produto( struct Produto *p ){
    scanf("%d", &p->codigo);
    scanf("%s", p->descricao);
    scanf("%f", &p->preco);
}
```

Exemplo 2: passagem por referência

```
struct Produto {
    int codigo;
    char descricao[20];
    float preco;
};

void le_produto( struct Produto *p ){
    scanf("%d", &p->codigo);
    scanf("%s", p->descricao);
    scanf("%f", &p->preco);
}

int main(){



}
```

Exemplo 2: passagem por referência

```
struct Produto {
    int codigo;
    char descricao[20];
    float preco;
};

void le_produto( struct Produto *p ){
    scanf("%d", &p->codigo);
    scanf("%s", p->descricao);
    scanf("%f", &p->preco);
}

int main(){
    struct Produto k;
    le_produto( &k );
    mostra_produto( k );
}
```

Considerações

- Dependendo do tamanho da estrutura (em bytes), é mais vantajoso sempre passar por referência;

Considerações

- Dependendo do tamanho da estrutura (em bytes), é mais vantajoso sempre passar por referência;

- Código fica mais rápido, pois não é preciso fazer uma cópia da estrutura a cada chamada de função;
  - Somente seu endereço é passado;

Exemplo 3: vetor de produtos

```
// Declaração da struct e das funções

int main(){
   struct Produto v[10];
   int i;









}
```

Exemplo 3: vetor de produtos

```
// Declaração da struct e das funções

int main(){
  struct Produto v[10];
  int i;
  for( i = 0 ; i < 10 ; i++ ){
    printf("Produto %d:\n", i + 1);

  }




}
```

Exemplo 3: vetor de produtos

```
// Declaração da struct e das funções

int main(){
  struct Produto v[10];
  int i;
  for( i = 0 ; i < 10 ; i++ ){
    printf("Produto %d:\n", i + 1);
    le_produto( &v[i] ); // ou v + i
  }




}
```

Exemplo 3: vetor de produtos

```
// Declaração da struct e das funções

int main(){
  struct Produto v[10];
  int i;
  for( i = 0 ; i < 10 ; i++ ){
    printf("Produto %d:\n", i + 1);
    le_produto( &v[i] ); // ou v + i
  }
  for( i = 0 ; i < 10 ; i++ ){
    printf("Dados do Produto %d:\n", i + 1);

  }
}
```

Exemplo 3: vetor de produtos

```
// Declaração da struct e das funções

int main(){
  struct Produto v[10];
  int i;
  for( i = 0 ; i < 10 ; i++ ){
    printf("Produto %d:\n", i + 1);
    le_produto( &v[i] ); // ou v + i
  }
  for( i = 0 ; i < 10 ; i++ ){
    printf("Dados do Produto %d:\n", i + 1);
    mostra_produto( v[i] ); // ou *(v + i)
  }
}
```

Exercício

- Modifique o exemplo 3 da seguinte forma:
  - Faça o vetor com alocação dinâmica (tal como exemplo da aula passada);
  - Implemente uma função de busca que recebe o vetor de produtos, a capacidade do vetor e o código do produto (*chave*);
    - Retorna o *índice* do produto no vetor (ou -1, caso não seja encontrado);

```
int busca(struct Produto *v, int n, int codigo);
```