

# Ponteiros em Structs

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados I (AE22CP)  
Engenharia de Computação  
Departamento Acadêmico de Informática (Dainf)  
Universidade Tecnológica Federal do Paraná (UTFPR)  
Campus Pato Branco

- Ponteiro de *Struct*
- Outros Tipos de Estruturas Heterogêneas
  - *Union*
  - *Enum*
- Erros Comuns em Ponteiros de Structs

- Tipos básicos de dados: `int`, `float`, `char`, ...
- Estrutura de dados homogêneas: vetores matrizes, strings, ...
- Estrutura de dados heterogêneas: *structs*, *unions* e *enums*
  - *struct*

```
struct nome_registro{  
    tipo1 campo1;  
    tipo2 campo2;  
    ...  
    tipoN campoN;  
};
```

- *struct*

```
typedef struct nome_estrutura nome_simplificado;
```

ou

```
typedef struct nome_estrutura{  
    ...  
}nome_simplificado;
```

ou

```
typedef struct{  
    ...  
}nome_simplificado;
```

- Operações com *struct*
  - Inicialização
  - Acesso aos elementos
  - Atribuição entre *structs*
- Vetores de *struct*
- Argumento e retorno de função *struct*
- Aninhamento de *structs*

```
typedef struct{  
    int algo;  
    Y y; // estrutura Y  
    Z z; // estrutura Z  
    ...  
}X;
```

## Ponteiro de *Struct*

- Assim como qualquer tipo de dado, o uso de ponteiros segue a mesma regra para as *structs*

```
typedef struct aluno{
    char nome[101];
    int RA;
    float coef;
}Aluno;

int main(void){
    Aluno variavel;
    Aluno *ponteiro;

    ponteiro = &variavel;

    return 0;
}
```

# Ponteiro de *Struct*

- Um ponteiro de *struct* recebe o endereço de uma *struct* do mesmo tipo
- Os operadores continuam o mesmo
  - & (endereço de)
  - \* (valor de)

```
int main(void){
    Aluno variavel;
    Aluno *ponteiro;

    ponteiro = &variavel;

    (*ponteiro).RA = 98765;
    (*ponteiro).coef = 0.57;
    strcpy((*ponteiro).nome, "Gil Away");

    return 0;
}
```



- Cuidado!
  - A expressão *\*var.campo* é equivalente a *(\*var.campo)*, mas tem significado muito diferente de *(\*var).campo*
    - O uso de *\*var.campo* e *(\*var.campo)* são usadas para variáveis de estrutura não declaradas como ponteiros, mas que possuem campos declarados como ponteiros
  - *(\*var).campo* é utilizada quando uma variável de estrutura é declarada como ponteiro

- Cuidado!
  - A expressão *\*var.campo* é equivalente a *(\*var.campo)*, mas tem significado muito diferente de *(\*var).campo*
    - O uso de *\*var.campo* e *(\*var.campo)* só é permitido para acessar campos de estrutura que foram declarados como ponteiros

```
typedef struct aluno{
    char nome[101];
    int *RA;
    float coef;
}Aluno;

int main(void){
    Aluno variavel;
    int x = 123;
    variavel.RA = &x;
    *variavel.RA = 321;
    return 0;
}
```

- Cuidado!
  - A expressão *\*var.campo* é equivalente a *(\*var.campo)*, mas tem significado muito diferente de *(\*var).campo*
    - *(\*var).campo* é usado para acessar conteúdo de um ponteiro de estrutura

```
int main(void){
    Aluno variavel;
    Aluno *ponteiro;

    int x;
    ponteiro = &variavel;
    (*ponteiro).RA = &x; // ou variavel.RA = &x
    (*ponteiro).coef = 0.57;
    *(*ponteiro).RA = 98765;
    return 0;
}
```

- Imprimir o endereço do registro

```
printf("%d", ponteiro);
```

- Imprimir o endereço de um campo do registro

```
printf("%d", &(*ponteiro).campo);
```

- Os ponteiros de *struct* possuem um operador para abreviar o comando de acesso ao valor
  - "->": acesso ao valor do campo no endereço
  - "p->" equivale ao uso de "(\*p)."

```
int main(void){
    Aluno variavel;
    Aluno *ponteiro;

    ponteiro = &variavel;

    ponteiro->RA = 12345;
    ponteiro->coef = 0.65;
    strcpy(ponteiro->nome, "Roberto");

    return 0;
}
```

- O operador "->" existe apenas para ponteiros de *structs*
- O uso de ponteiros de *struct* em funções é como qualquer ponteiro de variável nativa (int, char, etc)
  - Passagem por valor ou por referência
  - Retorno de função

```
typedef struct{  
    float x, y;  
}retangulo;
```

```
retangulo* soma(retangulo *r1, retangulo *r2){  
    retangulo aux;  
    retangulo *p_aux = &aux;  
    p_aux->x = r1->x + r2->x;  
    p_aux->y = r1->y + r2->y;  
    return p_aux; // Poderá não funcionar corretamente. Por  
    quê?  
}
```

## *Outros Tipos de Estruturas Heterogêneas*

# Outros Tipos de Estruturas Heterogêneas

## *Union*

- Uma união é um formato de dados que pode armazenar tipos diferentes, mas apenas um tipo de cada vez
- Uma *struct* pode armazenar um *int* e um *char* e um *double*
- Uma união pode armazenar um *int* ou um *char* ou um *double*

```
union uniao{
    tipo1 val;
    ...
    tipoN varN;
};

typedef uniao Uniao;
```

**ou**

```
typedef union{
    tipo1 val;
    ...
    tipoN varN;
}Uniao;
```



# Outros Tipos de Estruturas Heterogêneas

## *Union*

- Exemplo 1:

```
typedef union {
    int val_int;
    long val_long;
    double val_double;
}umpratodos;

int main() {
    umpratodos u;
    u.val_int = 15;
    printf("%d\n", u.val_int);
    u.val_double = 1.38;
    printf("%g\n", u.val_double);
    printf("%d\n", u.val_int);
    return 0;
}
```

# Outros Tipos de Estruturas Heterogêneas

## *Union*

- Exemplo 2:

```
typedef union {
    long nro_id;
    char char_id[20];
}identificador;

typedef struct {
    char marca[20];
    identificador id;
    int tipo;
}inventario;

int main() {
    inventario inv;
    strcpy(inv.marca, "Doli");
    inv.id.nro_id = 5678;
    return 0;
}
```

# Outros Tipos de Estruturas Heterogêneas

## Union

- Ponteiro de união possui a mesma sintaxe em relação à *struct*

```
int main() {
    umpratodos u;
    umpratodos *p_u;

    p_u = &u;
    u.val_int = 15;
    printf("%d\n", p_u->val_int);
    u.val_double = 1.38;
    printf("%g\n", p_u->val_double);
    printf("%d\n", p_u->val_int);
    return 0;
}
```

# Outros Tipos de Estruturas Heterogêneas

## *Union*

- União anônima

```
typedef struct {  
    char marca[20];  
  
    union{ // formato depende do tipo inventario  
        long nro_id; // inventários do tipo 1  
        char char_id[20]; // outros inventários  
    };  
    int tipo;  
}inventario;  
  
int main() {  
    inventario inv;  
    strcpy(inv.marca, "Doli");  
    inv.nro_id = 5678;  
    return 0;  
}
```

# Outros Tipos de Estruturas Heterogêneas

## *Enum*

- Define um enumerado como um novo tipo
- Estabelece símbolos (palavras) como constantes simbólicas para números inteiros entre 0 e o número de símbolos

```
enum enumerado {simb1, ..., simb1};
```

# Outros Tipos de Estruturas Heterogêneas

## *Enum*

- Exemplo:

```
enum espectro {vermelho, laranja, amarelo, verde, azul,  
violeta, anil, ultravioleta};
```

ou

```
typedef enum {vermelho, laranja, amarelo, verde, azul,  
violeta, anil, ultravioleta}espectro;
```

- Exemplo

```
espectro banda;

banda = azul; // válido, azul é um enumerador
banda = 2000; // inválido!, 2000 não é um enumerador
banda = laranja; // válido
banda++; // válido
banda = laranja + vermelho; // válido
...
int cor = azul; // válido, tipo espectro promovido a int
banda = 3; // válido, o tipo espectro atribui um valor para
cada tipo
cor = 3 + vermelho; // válido, vermelho é convertido para
int
...
banda = espectro(3); // inválido
```

# Outros Tipos de Estruturas Heterogêneas

## *Enum*

- Estabelecendo valores para enumeradores

```
enum bits {um = 1, dois = 2, quatro = 4, oito = 8};
```

```
enum grandepasso { primeiro, segundo = 100, terceiro};
```



- Escreva um código em C que armazene uma lista de pessoas, cada uma contendo:
  - Nome
  - Documento de identificação (RG ou CNH)
  - Sexo
  - Tipo sanguíneo
  - CPF
  - Data de nascimento
  - Endereço
- Escreva funções para:
  - Criar uma lista de contatos de tamanho  $n$
  - Preencher a lista de contatos
  - Imprimir a lista de contatos
  - Buscar um contato específico
  - Calcular a idade de uma pessoa para uma data específica

- Considere o seguinte exemplo de estrutura

```
typedef struct aluno{  
    char nome[101];  
    int RA;  
    float coef;  
}Aluno1;
```

# Erros Comuns em Ponteiros de *Structs*

```
int main(){  
    Aluno1 a;  
    Aluno1 *p;  
    p = a;  
    return 0;  
}
```

- Atribuição de uma variável em vez de endereço a um ponteiro
- Solução:

```
int main(){  
    Aluno1 a;  
    Aluno1 *p;  
    p = &a;  
    return 0;  
}
```

# Erros Comuns em Ponteiros de *Structs*

```
int main(){  
    Aluno1 a, b;  
    Aluno1 *p;  
    p = &b;  
    b = p;  
    return 0;  
}
```

- Atribuição de um endereço a uma variável do tipo Aluno1

- Solução:

```
int main(){  
    Aluno1 a, b;  
    Aluno1 *p;  
    p = &b;  
    b = *p;  
    return 0;  
}
```

# Erros Comuns em Ponteiros de *Structs*

```
int main(){
    Aluno1 a;
    Aluno1 *p;
    p = &b;
    *p.RA = 1234;
    return 0;
}
```

- O campo de um ponteiro de *struct* foi acessado inadequadamente
- Solução: usar  $(*p).RA$  ou  $p->RA$

```
int main(){
    Aluno1 a;
    Aluno1 *p;
    p = &b;
    p->RA = 1234;
    return 0;
}
```

# Erros Comuns em Ponteiros de *Structs*

```
int main(){
    Aluno1 a;
    Aluno1 *p;
    p = &b;
    p.RA = 1234;
    return 0;
}
```

- O campo de um ponteiro de *struct* foi acessado inadequadamente (2)
- Solução: usar  $(*p).RA$  ou  $p->RA$

```
int main(){
    Aluno1 a;
    Aluno1 *p;
    p = &b;
    p->RA = 1234;
    return 0;
}
```

- Considere o seguinte exemplo de estrutura

```
typedef struct aluno{  
    char nome[101];  
    int *RA;  
    float coef;  
}Aluno2;
```

# Erros Comuns em Ponteiros de *Structs*

```
int main(){  
    Aluno2 a;  
    a.RA = 10;  
    return 0;  
}
```

- Atribuição de um número inteiro a um ponteiro
- Solução: atribuição de um endereço por alocação dinâmica (assunto da próxima aula) ou vincular um endereço de variável

```
int main(){  
    Aluno2 a;  
    int x = 10;  
    a.RA = &x;  
    return 0;  
}
```



# Erros Comuns em Ponteiros de *Structs*

```
int main(){
    Aluno2 a;
    int x;
    a.RA = &x;
    a.RA = 10;
    return 0;
}
```

- Atribuição de um número inteiro a um campo ponteiro sem o uso do operador \*
- Solução: usar o operador \*

```
int main(){
    Aluno2 a;
    int x;
    a.RA = &x;
    *a.RA = 10;
    return 0;
}
```

# Erros Comuns em Ponteiros de *Structs*

```
int main(){
    Aluno2 a;
    Aluno2 *p;
    int x;
    p = &b;
    *p.RA = &x;
    return 0;
}
```

- Foi utilizado o operador `*` na forma de "valor de" para atribuição de endereço
- Solução: usar `(*p).RA` ou `p->RA`

```
int main(){
    Aluno2 a;
    Aluno2 *p;
    int x;
    p = &b;
    p->RA = &x;
    return 0;
}
```

# Erros Comuns em Ponteiros de *Structs*

```
int main(){
    Aluno2 a;
    Aluno2 *p;
    int x;
    p = &b;
    p->RA = &x;
    *p.RA = 10;
    return 0;
}
```

- Em vez de utilizar "->" em um ponteiro de *struct* para o acesso a um campo, foi utilizado "."
- Solução: usar *\*(p).RA* ou *\*p->RA*

```
int main(){
    Aluno2 a;
    Aluno2 *p;
    int x;
    p = &b;
    p->RA = &x;
    *p->RA = 10;
    return 0;
}
```



Arakaki, R.; Arakaki, J.; Angerami, P. M.; Aoki, O. L.; Salles, D. S.

*Fundamentos de programação C: técnicas e aplicações.*

LTC, 1990.



Deitel, H. M.; Deitel, P. J.

*Como programar em C.*

LTC, 1999.



Pereira, S. L.

*Estrutura de Dados e em C: uma abordagem didática.*

Saraiva, 2016.



Tenenbaum, A.; Langsam, Y.

*Estruturas de Dados usando C.*

Pearson, 1995.