Programação de Computadores

Raoni F. S. Teixeira

Aula 14 - Registros e Outras Estruturas

1 Introdução

Como podemos armazenar todos os dados de um aluno: RA, telefone, nome, endereço e data de nascimento?

Uma primeira abordagem seria usar variáveis separadas:

```
int ra;
int telefone;
char nome[30];
char endereco[100];
int dia_nascimento;
int mes_nascimento;
int ano_nascimento;

scanf("%d%d%s%s%d%d%d", &ra, &telefone, nome, endereco,
      &dia_nascimento, &mes_nascimento, &ano_nascimento);
```

Esta solução funciona, mas apresenta vários problemas práticos.

1.1 Problema 1: Copiar Dados

Se quiséssemos copiar os dados do aluno, precisaríamos:

```
int ra, copia_ra;
int telefone, copia_telefone;
char nome[30], copia_nome[30];
char endereco[100], copia_endereco[100];
int dia_nascimento, copia_dia_nascimento;
int mes_nascimento, copia_mes_nascimento;
int ano_nascimento, copia_ano_nascimento;
int i;

copia_ra = ra;
copia_telefone = telefone;
```

```
for (i = 0; nome[i] != '\0'; i++)
    copia_nome[i] = nome[i];
copia_nome[i] = '\0';
// ... copiar endereco da mesma forma
copia_dia_nascimento = dia_nascimento;
copia_mes_nascimento = mes_nascimento;
copia_ano_nascimento = ano_nascimento;
```

Muito trabalhoso! E ainda nem copiamos o endereço.

1.2 Problema 2: Passar Dados para Funções

Para passar os dados como parâmetros de uma função, a assinatura ficaria extremamente longa:

```
void imprimir_dados(int ra, int telefone, char nome[],
                    char endereco[], int dia_nascimento,
                    int mes_nascimento, int ano_nascimento) {
    // ...
}

int main() {
    // ...
    imprimir_dados(ra, telefone, nome, endereco,
                   dia_nascimento, mes_nascimento, ano_nascimento);
    // ...
}
```

Imagine adicionar mais campos (e-mail, CPF, curso, etc.)! A lista de parâmetros ficaria inviável.

1.3 A Solução: Registros

E se pudéssemos agrupar todos esses dados relacionados em uma única estrutura? Com registros (*structs*), podemos fazer exatamente isso:

```
void imprimir_dados(struct ficha_aluno ficha) {
    // ...
}

int main() {
    struct ficha_aluno ficha, copia_ficha;

    // Lendo
    ficha = ler_ficha();

    // Copiando - uma unica atribuicao!
    copia_ficha = ficha;
```

```
    // Passando por parametro - apenas um argumento!
    imprimir_dados(ficha);

    return 0;
}
```

Muito mais simples e organizado!

2 Registros

2.1 Definição

Um **registro** (*struct* em C) é uma coleção de variáveis relacionadas de vários tipos, organizadas em uma única estrutura e referenciadas por um nome comum.

Características dos registros:

- Cada variável dentro do registro é chamada de **membro** (ou campo).
- Cada membro é acessado por um nome específico usando o operador ponto (`.`).
- Cada estrutura define um novo tipo, com as mesmas características de um tipo padrão da linguagem.
- Membros podem ser de tipos diferentes (ao contrário de vetores).
- Registros podem conter outros registros (estruturas aninhadas).

2.2 Declaração de Estruturas

A declaração de uma estrutura segue a sintaxe:

```
struct identificador {
    tipo1 membro1;
    tipo2 membro2;
    ...
    tipoN membroN;
};
```

Importante: Note o ponto e vírgula após a chave de fechamento! É obrigatório.

2.3 Declaração de Variáveis do Tipo Estrutura

Após definir a estrutura, podemos declarar variáveis (registros) desse tipo:

```
struct identificador nome_registro;
```

Em C:

- Declaramos o *tipo* de uma estrutura apenas uma vez (geralmente antes do `main`).
- Podemos declarar várias variáveis (registros) da mesma estrutura.

2.4 Exemplo: Ficha de Aluno

```
struct ficha_aluno {
    int ra;
    int telefone;
    char nome[30];
    char endereco[100];
    int dia_nascimento;
    int mes_nascimento;
    int ano_nascimento;
};

// Declarando variaveis do tipo ficha_aluno
struct ficha_aluno aluno1, aluno2;
```

2.5 Estruturas Aninhadas

As variáveis `dia_nascimento`, `mes_nascimento` e `ano_nascimento` são relacionadas. Podemos agrupá-las em uma estrutura separada:

```
struct data {
    int dia;
    int mes;
    int ano;
};

struct ficha_aluno {
    int ra;
    int telefone;
    char nome[30];
    char endereco[100];
    struct data nascimento;  // Estrutura aninhada
};
```

Esta organização é mais clara e modular. Podemos reutilizar a estrutura `data` em outros contextos.

3 Usando Registros

3.1 Acessando Membros

Para acessar um membro de um registro, usamos o operador ponto (.):

```
registro.membro
```

3.2 Exemplos de Acesso

```
struct ficha_aluno aluno;

// Atribuindo valores
aluno.ra = 123456;
aluno.telefone = 987654321;
strcpy(aluno.nome, "Maria␣Silva");

// Lendo valores
printf("Nome:␣%s\n", aluno.nome);
printf("RA:␣%d\n", aluno.ra);
```

Para estruturas aninhadas, usamos múltiplos pontos:

```
struct ficha_aluno aluno;

// Atribuindo data de nascimento
aluno.nascimento.dia = 15;
aluno.nascimento.mes = 8;
aluno.nascimento.ano = 2000;

// Imprimindo aniversario
printf("Aniversario:␣%d/%d/%d\n",
       aluno.nascimento.dia,
       aluno.nascimento.mes,
       aluno.nascimento.ano);
```

3.3 Exemplo Completo: Média das Notas

```
#include <stdio.h>
#define NUM_ALUNOS 60

struct ficha_notas {
    int ra;
    float p1, p2;
};
```

```
int main() {
    struct ficha_notas notas[NUM_ALUNOS];
    float soma = 0;
    int i;

    // Lendo dados dos alunos
    for (i = 0; i < NUM_ALUNOS; i++) {
        printf("Digite o RA e as notas P1 e P2: ");
        scanf("%d%f%f", &notas[i].ra, &notas[i].p1, &notas[i].p2);
    }

    // Calculando media
    for (i = 0; i < NUM_ALUNOS; i++)
        soma += notas[i].p1 + notas[i].p2;

    printf("Media da nota final: %.2f\n", soma / (2 * NUM_ALUNOS));

    return 0;
}
```

Note como criamos um *vetor de registros*: `notas[NUM_ALUNOS]`. Cada elemento do vetor é um registro completo com três campos.

4 Registros em Funções

Registros podem ser usados como qualquer outro tipo em funções:

- Podem ser passados como parâmetros (por valor ou por referência).
- Podem ser retornados por funções.
- Por padrão, são passados **por valor** (uma cópia é criada).

4.1 Passagem por Valor

```
#include <stdio.h>

struct retangulo {
    char cor[10];
    float largura, altura;
};

struct retangulo ler_retangulo() {
    struct retangulo ret;
    printf("Digite largura, altura e cor: ");
```

```
    scanf("%f%f%s", &ret.largura, &ret.altura, ret.cor);
    return ret;
}

float area_retangulo(struct retangulo ret) {
    return ret.largura * ret.altura;
}

int main() {
    struct retangulo ret;
    ret = ler_retangulo();
    printf("A area e %.2f\n", area_retangulo(ret));
    return 0;
}
```

4.2 Passagem por Referência

Para modificar um registro dentro de uma função, devemos passá-lo por referência (usando ponteiros):

```
#include <stdio.h>

struct retangulo {
    char cor[10];
    float largura, altura;
};

void girar_retangulo(struct retangulo *ret) {
    float aux;
    aux = (*ret).largura;
    (*ret).largura = (*ret).altura;
    (*ret).altura = aux;
}

int main() {
    struct retangulo ret = {"azul", 10.0, 5.0};

    printf("Antes: %.1f x %.1f\n", ret.largura, ret.altura);
    girar_retangulo(&ret);
    printf("Depois: %.1f x %.1f\n", ret.largura, ret.altura);

    return 0;
}
```

Importante: Devemos usar parênteses em `(*ret).largura` porque o operador `.` tem maior precedência que `*`. Sem parênteses, o compilador tentaria interpretar como `*(ret.largura)`, o que causaria erro.

4.3 Notação Alternativa: Operador Seta

Como acessar membros de ponteiros para estruturas é muito comum, C fornece uma notação mais conveniente: o operador seta (`->`):

```
void girar_retangulo(struct retangulo *ret) {
    float aux;
    aux = ret->largura;
    ret->largura = ret->altura;
    ret->altura = aux;
}
```

As notações `(*ret).largura` e `ret->largura` são completamente equivalentes. A segunda é mais limpa e é a preferida pelos programadores C.

Resumo dos operadores:

- `registro.membro` — acessa membro de um registro
- `(*ponteiro).membro` — acessa membro via ponteiro
- `ponteiro->membro` — notação equivalente mais limpa

5 Enumerações

Como representar um conjunto conhecido e finito de elementos, como os meses do ano ou um conjunto de cores?

5.1 Solução com #define

Uma primeira abordagem seria usar `#define`:

```
#define VERMELHO 0
#define VERDE    1
#define AZUL     2
#define MARROM   3
#define ROXO     4
#define AMARELO  5
```

Esta solução funciona, mas é trabalhosa e propensa a erros (podemos acidentalmente atribuir o mesmo número a duas constantes).

5.2 Enumerações em C

C fornece o tipo `enum` para representar conjuntos enumerados:

```
enum cores {
    VERMELHO,
    VERDE,
    AZUL,
    MARROM,
    ROXO,
    AMARELO
};
```

Uma enumeração atribui automaticamente um inteiro a cada constante, começando de 0:

- `VERMELHO` vale 0
- `VERDE` vale 1
- `AZUL` vale 2
- E assim por diante...

5.3 Usando Enumerações

Podemos usar as constantes enumeradas como qualquer inteiro:

```
enum cores cor1 = AZUL;
int cor2 = VERDE;

printf("Azul␣e␣uma␣cor␣");
switch (cor1) {
    case AZUL: case VERMELHO: case AMARELO:
        printf("primaria.\n");
        break;
    default:
        printf("derivada.\n");
}

printf("O␣codigo␣do␣verde␣e␣%d\n", cor2);
```

Saída:

```
Azul e uma cor primaria.
O codigo do verde e 1
```

5.4 Mudando o Valor Inicial

Para mudar o valor inicial ou atribuir valores específicos, use o operador =:

```
enum meses {
    JANEIRO = 1,    // Comeca em 1, nao 0
    FEVEREIRO,      // 2
    MARCO,          // 3
    ABRIL,          // 4
    MAIO,           // 5
    JUNHO,          // 6
    JULHO,          // 7
    AGOSTO,         // 8
    SETEMBRO,       // 9
    OUTUBRO,        // 10
    NOVEMBRO,       // 11
    DEZEMBRO        // 12
};
```

Apenas o primeiro valor precisa ser especificado; os demais são incrementados automaticamente.

5.5 Exemplo com Estruturas

Enumerações são frequentemente usadas em conjunto com estruturas:

```
enum tipo_forma {
    CIRCULO,
    RETANGULO,
    TRIANGULO
};

struct forma {
    enum tipo_forma tipo;
    float area;
    char cor[20];
};

int main() {
    struct forma f;
    f.tipo = RETANGULO;
    f.area = 25.5;
    strcpy(f.cor, "azul");

    if (f.tipo == RETANGULO)
        printf("E␣um␣retangulo!\n");

    return 0;
}
```

6 Typedef: Apelidos de Tipos

6.1 Motivação

Representamos dados usando tipos primitivos (`int`, `float`, etc.), mas às vezes:

- Dados diferentes têm o mesmo tipo (nota e altura são ambos `float`)
- Queremos nomes mais descritivos para estruturas e enumerações
- Escrever `struct nome_estrutura` repetidamente é verboso

6.2 O Comando typedef

O comando `typedef` cria apelidos (sinônimos) para tipos existentes:

```
typedef tipo_existente Apelido;
```

6.3 Exemplos de Typedef

```
// Apelido para tipo primitivo
typedef float Medida;

// Apelido para enumeracao
typedef enum {
    VERMELHO,
    VERDE,
    AZUL
} Cor;

// Apelido para estrutura
typedef struct {
    Medida altura, largura;
    Cor cor_face, cor_linha;
} Retangulo;

int main() {
    Retangulo ret;  // Nao precisa mais escrever "struct"!
    ret.cor_linha = VERMELHO;
    ret.largura = 7.5;
    ret.altura = 10.0;

    Medida area = ret.largura * ret.altura;
    printf("Area: %.2f\n", area);

    return 0;
```

```
}
```

Note como o código fica mais limpo: usamos `Retangulo` em vez de `struct retangulo`, e `Cor` em vez de `enum cores`.

6.4 Typedef com Estruturas Nomeadas

Também podemos usar `typedef` com estruturas que têm nome:

```
typedef struct ponto {
    float x, y;
} Ponto;

// Agora podemos usar tanto:
struct ponto p1;    // Forma tradicional
Ponto p2;           // Usando o typedef (preferido)
```

Convenção: É comum usar nomes com letra maiúscula para tipos definidos com `typedef`, para distingui-los de variáveis e funções.

7 Exercícios

1. Escreva uma estrutura `ponto` para representar um ponto no plano cartesiano (coordenadas x e y).
2. Escreva uma estrutura `triangulo` que contém três pontos correspondentes aos três vértices.
3. Escreva uma função que receba dois pontos e retorne a distância euclidiana entre eles:
$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$
4. Escreva uma função que receba um triângulo e devolva o perímetro (soma das distâncias entre os vértices).
5. Escreva uma função que receba um triângulo e devolva a área usando a fórmula de Heron:
$$A = \sqrt{s(s-a)(s-b)(s-c)}$$
onde $s = \frac{a+b+c}{2}$ é o semiperímetro.
6. Escreva uma função que desloque um triângulo à direita de uma distância d. A função deve modificar o triângulo original (passagem por referência).

7. Escreva uma função que retorne o tipo de um triângulo (escaleno, isósceles ou equilátero). Use uma enumeração:

```
enum tipo_triangulo {
    ESCALENO,
    ISOSCELES,
    EQUILATERO
};
```

8. Crie uma estrutura `aluno` com nome, RA, e um vetor de 4 notas. Escreva funções para:
   - Ler os dados de um aluno
   - Calcular a média das notas
   - Imprimir os dados formatados
9. Crie uma estrutura `circulo` com centro (um ponto) e raio. Escreva funções para calcular:
   - Área do círculo
   - Circunferência
   - Verificar se um ponto está dentro do círculo
10. Implemente um sistema de gerenciamento de formas geométricas que suporte círculos, retângulos e triângulos:

```
enum tipo_forma {
    CIRCULO, RETANGULO, TRIANGULO
};

typedef struct {
    enum tipo_forma tipo;
    union {
        struct { float raio; } circ;
        struct { float base, altura; } ret;
        struct { float lado1, lado2, lado3; } tri;
    } dados;
} Forma;
```

    Escreva funções para calcular área e perímetro de qualquer forma.

11. Implemente um jogo de cartas usando estruturas:

```
enum naipe { COPAS, ESPADAS, OUROS, PAUS };
enum valor { AS = 1, DOIS, TRES, /* ... */ REI = 13 };

typedef struct {
```

```
    enum naipe naipe;
    enum valor valor;
} Carta;

typedef struct {
    Carta cartas[52];
    int topo;
} Baralho;
```

Implemente funções para embaralhar, distribuir cartas, etc.

8 Resumo

Pontos-chave desta aula:

- Registros (`struct`) agrupam variáveis relacionadas de tipos diferentes
- Declaração: `struct nome { membros };`
- Acesso a membros: `registro.membro`
- Estruturas podem ser aninhadas
- Registros são passados por valor por padrão
- Use ponteiros para modificar registros em funções
- Operador seta (`->`): `ponteiro->membro` = `(*ponteiro).membro`
- Enumerações (`enum`) definem conjuntos de constantes inteiras
- `typedef` cria apelidos para tipos, tornando o código mais legível