

Registros – *Structs*

CSI030 – PROGRAMAÇÃO DE COMPUTADORES I



Links para os conteúdos

- Videoaulas UFJF (Aula 9 – partes de 1 a 9):
<https://youtube.com/playlist?list=PL1K9y5L0Vn9UkTDvSDDn3CzJFftFru1a6>
- Slides adicionais:
<https://drive.google.com/file/d/1kcbN8dzjSu12wmfxlnMTXJiqSWL6qw7b/view?usp=sharing>

Agradecimentos

- Agradecemos aos Professores do Departamento de Ciência da Computação da UFJF que gentilmente permitiram a utilização das videoaulas e dos conteúdos elaborados por eles.



UFOP

Universidade Federal
de Ouro Preto

Registros - *Structs*

- Até o momento foram abordadas apenas estruturas de dados **homogêneas**:
 - Vetores, matrizes e strings.
- Com tipos de dados primitivos:
 - Int, float, char, ...

Registros - *Structs*

- Por exemplo, podemos armazenar as três notas de uma turma com 40 pessoas numa matriz:
 - **float** notas[40][3];
- Para o nome, poderíamos criar outra matriz:
 - **char** nomes[40][100];



Registros - *Structs*

- E para os demais dados, como matrícula, curso, gênero, dentre outros?
 - Criar uma matriz para cada um deles ficaria inviável de se manipular.
 - Ao adicionar ou excluir uma pessoa, todas as variáveis precisariam ser acessadas e atualizadas.
- E se pudéssemos agrupar esses dados?



UFOP

Universidade Federal
de Ouro Preto

Registros - *Structs*

- Os dados podem combinados em variáveis compostas.
- Essas variáveis são **heterogêneas**.
- Elas são conhecidas como registros ou estruturas (em C, são definidas como **structs**).



Registros - *Structs*

- Um **registro (struct)** é a maneira de se agrupar variáveis em C, sejam elas de mesmo tipo ou de tipos diferentes.
- As variáveis unidas em um registro se relacionam de modo a criar um contexto maior.
- **Exemplos** de uso de registros:
 - Registro de Alunos: armazena nome, matrícula, médias, faltas.
 - Registro de Pacientes: armazena nome, endereço, convênio, histórico hospitalar.
 - Registro de funcionários: armazena nome, endereço, cargo, salário.

Declaração do registro

- Um registro é declarado da seguinte maneira:

```
struct nome_tipo_registro {  
    tipo_1 variavel_1;  
    tipo_2 variavel_2;  
    ...  
    tipo_n variavel_n;  
};
```

- Uma variável deste registro deve ser declarada da seguinte maneira:

```
struct nome_tipo_registro variavel_registro;
```



UFOP

Universidade Federal
de Ouro Preto

Registros - *Structs*

- Um registro define um **novo tipo de dados** com nome **nome_tipo_registro** que possui os campos **variavel_i**.
- O campo **variavel_i** é uma variável do tipo **tipo_i** e será acessível a partir de uma variável do novo tipo **nome_tipo_registro**.
- **variavel_registro** é uma variável do tipo **nome_tipo_registro** e possui, internamente, os campos **variavel_i** do tipo declarado.
- A **declaração de um registro** pode ser feita dentro de uma sub-rotina ou fora dela.

Registros - *Structs*

- Um registro define um **novo tipo de dados** com nome **nome_tipo_registro** que possui os campos **variavel_i**.
- O campo **variavel_i** é uma variável do tipo **tipo_i** e será acessível a partir de uma variável do novo tipo **nome_tipo_registro**.
- **variavel_registro** é uma variável do tipo **nome_tipo_registro** e possui, internamente, os campos **variavel_i** do tipo declarado.
- A **declaração de um registro** pode ser feita dentro de uma sub-rotina ou fora dela.



UFOP

Universidade Federal
de Ouro Preto

Exemplo de utilização

```
// Definição da estrutura
struct Aluno {
    char nome[50];
    int idade;
    char genero;
    int turma;
};

int main(void){
    // Declaração de variáveis
    struct Aluno aluno1, aluno2;
}
```

Acesso aos campos do registro

- Os campos de um registro podem ser acessados individualmente a partir de variáveis do tipo do registro da seguinte maneira:

`variavel_registro.variavel_i`

- Cada campo **`variavel_registro.variavel_i`** se comporta como uma variável do **tipo** do campo, ou seja, **`tipo_i`**.
- Isso significa que todas as operações válidas para variáveis do tipo **`tipo_i`** são válidas para o campo acessado por **`variavel_registro.variavel_i`**.

Inicialização de registros

- Na inicialização de registros, os campos obedecem à ordem de declaração:

```
struct contaBancaria {  
    int numero;  
    char idCorrentista[15];  
    float saldo;  
};  
  
struct contaBancaria conta = {1, "MG1234567", 100.0};
```

Atribuição de registros

- Pode-se copiar o conteúdo de uma estrutura para outra utilizando uma atribuição simples.

```
struct ponto {int x; int y;};
```

```
int main() {
```

```
    struct ponto p = {1,2}, q;
```

```
    q.x = 3; q.y = 4;
```

```
    q = p;
```

```
    q.x = 5;
```

```
}
```

p		q	
x	y	x	y
1	2		
1	2	3	4
1	2	1	2
1	2	5	2

Vetores de registros

- **Vetores de registros** podem ser criados da mesma maneira que se criam vetores de tipos primitivos (**int**, **float**, **char**, **double**).
- É necessário definir a estrutura antes de declarar o vetor.

```
struct ponto {int x; int y;};
```

```
int main() {  
    struct ponto v[2];  
    v[0].x = 3; v[0].y = 4;  
    v[1].x = 5; v[1].y = 6;  
}
```


Registros como parâmetros de sub-rotinas

- Pode-se passar um registro como parâmetro de sub-rotinas.

```
struct ponto {int x; int y;};  
void imprimePonto( struct ponto p ){  
    printf("Coordenadas (%d, %d)", p.x, p.y);  
}  
int main() {  
    struct ponto p = {1,2};  
    imprimePonto(p);  
}
```



Sinônimos de tipos

- O comando **typedef** permite dar novos nomes (sinônimos) a tipos de dados existentes.

```
typedef <tipo> <sinônimo>
```

- Por exemplo:

```
typedef int inteiro;  
typedef char character;
```

- Para declarar as variáveis a partir dos sinônimos, temos:

```
inteiro idade;  
character genero;
```

Sinônimos de tipos

- Assim, podemos fazer isso também com os registros e simplificar a declaração.

```
struct ponto {int x; int y;};
```

```
typedef ponto Ponto;
```

```
Ponto r, s;
```



UFOP

Universidade Federal
de Ouro Preto

Sinônimos de tipos

- Outro exemplo:

```
struct contabancaria {  
    inteiros numero;  
    caracteres idCorrentista[15];  
    float saldo;  
};  
typedef struct contabancaria CB;  
  
CB conta1, conta2;
```

Sinônimos de tipos

- Podemos ainda associar uma definição de **struct** a um novo tipo de dados, evitando a repetição da palavra **struct** pelo código:

```
typedef struct contabancaria {  
    inteiros numero;  
    caracteres idCorrentista[15];  
    float saldo;  
} CB;
```

```
CB conta1, conta2;
```

Alocação dinâmica de Registros

- Podemos também utilizar alocação dinâmica de memória com ponteiros de registros.
- Para acessar os elementos de um registro, deve-se acessar o registro e, depois, os elementos;

```
typedef struct ponto {int x; int y;} Ponto;  
Ponto *p;  
p = (Ponto*) malloc (sizeof(Ponto));  
(*p).x = 3; // Os parênteses são obrigatórios porque a precedência do  
(*p).y = 4; // operador * é menor que o do operador . (ponto)  
free(p);
```



Alocação dinâmica de Registros

- O operador -> facilita a utilização dos registros alocados dinamicamente, permitindo acessar diretamente o conteúdo de um campo do registro.

```
typedef struct ponto {int x; int y;} Ponto;  
Ponto *p;  
p = (Ponto*) malloc (sizeof(Ponto));  
p->x = 3;  
p->y = 4;  
free(p);
```



UFOP

Universidade Federal
de Ouro Preto

Exercícios





UFOP

Universidade Federal
de Ouro Preto

Exercício 1

- Crie uma estrutura para representar um grupo de alunos. Cada registro deve possuir o campo nome e três notas. Solicite que o usuário preencha as informações e, em seguida, imprima todos os dados dos respectivos alunos.



Exercício 2

- Crie uma estrutura para representar um grupo de alunos. Desta vez, solicite que o usuário informe quantos alunos devem ser criados e aloque-os dinamicamente. Cada registro deve possuir o campo nome e três notas. Solicite que o usuário preencha as informações e, em seguida, imprima todos os dados dos respectivos alunos.



UFOP

Universidade Federal
de Ouro Preto

Exercício 3

- Crie uma estrutura ponto contendo os campos x e y, coordenadas de um ponto dadas por valores reais.
- Declare as variáveis p1 e p2 (dois pontos), preencha os campos para cada variável, via teclado, e imprima a distância euclidiana entre p1 e p2.



Referências

- DEITEL, P; DEITEL, H. C How to Program. 6a Ed. Pearson, 2010.
- Material de aula do Prof. Ricardo Anido, da UNICAMP:
<http://www.ic.unicamp.br/~ranido/mc102/>
- Material de aula da Profa. Virgínia F. Mota:
<https://sites.google.com/site/virginiaferm/home/disciplinas>
- Conteúdos disponibilizados pelos Professores do Departamento de Ciência da Computação da UFJF: <https://sites.google.com/site/algoritmosufjf/ere/turma-a-a-j/material-a-a-j-ere>