

Programação Estruturada

Registros (*structs*)

Professores Emílio Francesquini e Carla Negri Lintzmayer

2018.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC



Registros

Um registro é um mecanismo da linguagem C para agrupar várias variáveis, que inclusive podem ser de tipos diferentes, mas que dentro de um contexto, fazem sentido estarem juntas.

- Exemplos de uso de registros:
 - Registro de alunos para guardar os dados: nome, RA, médias de provas, médias de labs, etc. . .
 - Registro de pacientes para guardar os dados: nome, endereço, histórico de doenças, etc. . .

Declarando um novo tipo de registro

- Para criarmos um novo tipo de registro usamos a palavra chave **struct** da seguinte forma:

```
1 struct nome_registro {
2     tipo_1 nome_campo_1;
3     tipo_2 nome_campo_2;
4     tipo_3 nome_campo_3;
5     ...
6     tipo_n nome_campo_n;
7 };
```

- Cada **nome_campo_i**, é um identificador que será do tipo **tipo_i** (são declarações de variáveis simples).

Declarando um novo tipo de registro

Exemplo:

```
1 struct Aluno {  
2     char nome[80];  
3     float nota;  
4 }; /* estamos criando um novo tipo "struct Aluno" */
```

Declarando um novo tipo de registro

A declaração do registro pode ser feita dentro de uma função ou fora dela. Usualmente, ela é feita fora de qualquer função, para que qualquer função possa usar dados do tipo de registro criado.

```
1  #include <stdio.h>
2
3  /* Declare tipos registro aqui */
4
5  int main() {
6      /* Comandos */
7  }
```

Declarando um registro

A próxima etapa é declarar uma variável do tipo **struct nome_registro**, que será usada dentro de seu programa, como no exemplo abaixo:

```
1  #include <stdio.h>
2
3  struct Aluno {
4      char nome[80];
5      float nota;
6  };
7
8  int main() {
9      /* variáveis são do tipo "struct Aluno" */
10     struct Aluno a, b;
11     ...
12 }
```

- Podemos acessar individualmente os campos de uma determinada variável registro como se fossem variáveis normais. A sintaxe é:

```
1 variável_registro.nome_do_campo
```

- Os campos individuais de uma variável registro tem o mesmo comportamento de qualquer variável do tipo do campo.

Utilizando os campos de um registro

```
1 struct Aluno {
2     char nome[45];
3     float nota;
4 };
5
6 int main() {
7     /* variáveis do tipo "struct Aluno" */
8     struct Aluno a, b;
9     a.nota = 4.7;
10    b.nota = 2 * a.nota;
11    return 0;
12 }
```

Utilizando os campos de um registro

```
1  #include <stdio.h>
2  #include <string.h>
3  struct Aluno {
4      char nome[45];
5      float nota;
6  };
7  int main() {
8      struct Aluno a, b;
9
10     strcpy(a.nome, "Helen");
11     a.nota = 8.6;
12     strcpy(b.nome, "Dilbert");
13     b.nota = 8.2;
14     printf("a.nome = %s, a.nota = %f\n", a.nome, a.nota);
15     printf("b.nome = %s, b.nota = %f\n", b.nome, b.nota);
16     return 0;
17 }
```

Lendo e escrevendo registros

- A leitura de um registro a partir do teclado deve ser feita campo a campo, como se fossem variáveis independentes.
- A mesma coisa vale para a escrita, que deve ser feita campo a campo.

```
1 struct Aluno a, b;
2
3 printf("Digite o nome:");
4 fgets(a.nome, 80, stdin);
5 printf("Digite a nota:");
6 scanf("%f", &a.nota); getchar();
7
8 printf("Digite o nome:");
9 fgets(b.nome, 80, stdin);
10 printf("Digite a nota:");
11 scanf("%f", &b.nota); getchar();
12
13 printf("a.nome = %s, a.nota = %.2f\n", a.nome, a.nota);
14 printf("b.nome = %s, b.nota = %.2f\n", b.nome, b.nota);
```

- Podemos atribuir um registro a outro diretamente se eles forem do mesmo tipo:

```
var1_registro = var2_registro;
```

- Automaticamente é feita uma cópia de cada campo de **var2_registro** para **var1_registro**.

Atribuição de registros: exemplo

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct Aluno {
5      char nome[80];
6      float nota;
7  };
8  int main() {
9      struct Aluno a, b;
10
11     printf("Digite o nome:");
12     fgets(a.nome, 80, stdin);
13     printf("Digite a nota:");
14     scanf("%f", &a.nota); getchar();
15
16     b = a; /* Atribuição de registros */
17
18     printf("b.nome = %s, b.nota = %.2f\n", b.nome, b.nota);
19 }
```

- A declaração e uso de vetores de registros se dá da mesma forma que vetores dos tipos básicos vistos anteriormente.
 - Para declarar:

```
1 struct Aluno turma[60];
```

- Para usar:

```
1 turma[indice].campo;
```

Exemplo de vetor de registros

```
1  #include <stdio.h>
2  #include <string.h>
3  struct Aluno {
4      char nome[80];
5      float nota;
6  };
7  int main() {
8      struct Aluno turma[5];
9      int i;
10     float media = 0;
11     for (i = 0; i < 5; i++) {
12         printf("Digite o nome:");
13         fgets(turma[i].nome, 80, stdin);
14         printf("Digite a nota:");
15         scanf("%f", &turma[i].nota); getchar();
16     }
17     for (i = 0; i < 5; i++)
18         media = media + turma[i].nota;
19     printf("Media da turma = %.2f\n", media / 5.0);
20     return 0;
21 }
```

- Registros podem ser usados tanto como parâmetros em funções quanto como em retorno de funções.
- Neste caso o comportamento de registros é similar ao de tipos básicos.

Vamos criar as seguintes funções:

-
- `struct Aluno leAluno();`
-

Esta função faz a leitura dos dados de um registro **struct Aluno** e devolve o registro lido.

- `void imprimeAluno(struct Aluno a);`
-

Esta função recebe como parâmetro um registro **struct Aluno** e imprime os dados do registro.

- `void listarTurma(struct Aluno turma[], int n);`
-

Esta função recebe como parâmetros um vetor do tipo **struct Aluno** representando uma turma, e um inteiro **n** indicando o tamanho do vetor e imprime os dados de todos os alunos.

Implementação das funções:

```
1  struct Aluno leAluno() {
2      struct Aluno aluno;
3
4      printf("Digite o Nome: ");
5      fgets(aluno.nome, 80, stdin);
6      printf("Digite a Nota: ");
7      scanf("%f", &aluno.nota); getchar();
8
9      return aux;
10 }
```

```
1 void imprimeAluno(struct Aluno a) {
2     printf("Dados de um aluno --- ");
3     printf("Nome: %s. Nota: %.2f\n", a.nome, a.nota);
4 }
5
6 void listarTurma(struct Aluno turma[], int n) {
7     int i;
8     printf("Imprimindo a turma\n");
9     for (i = 0; i < n; i++)
10         imprimeAluno(turma[i]);
11 }
```

Com as funções implementadas podemos criar o seguinte exemplo de programa.

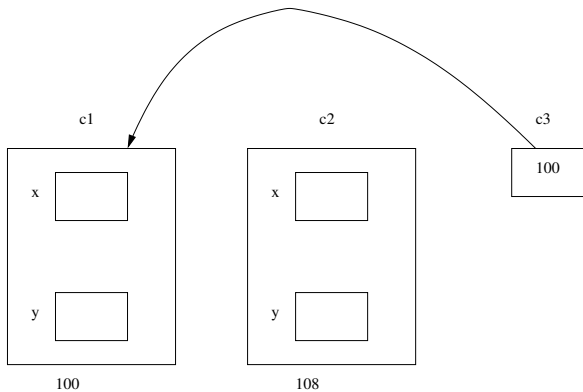
```
1  #include <stdio.h>
2  #include <string.h>
3  #define MAX 4
4  struct Aluno {
5      char nome[80];
6      float nota;
7  };
8  struct Aluno leAluno();
9  void imprimeAluno(struct Aluno a);
10 void listarTurma(struct Aluno turma[], int n);
11 int main() {
12     int i;
13     struct Aluno turma[MAX];
14     for (i = 0; i < MAX; i++)
15         turma[i] = leAluno();
16     listarTurma(turma, MAX);
17     return 0;
18 }
```

Ponteiros e registros

- Ao criarmos uma variável de um tipo **struct**, esta é armazenada na memória como qualquer outra variável, e portanto possui um endereço.
- É possível então criar um ponteiro para uma variável de um tipo **struct**!

```
1  #include <stdio.h>
2
3  struct Coordenada {
4      double x;
5      double y;
6  };
7
8  int main() {
9      struct Coordenada c1, c2, *c3;
10     c3 = &c1;
11     return 0;
12 }
```

Ponteiros e registros



O que será impresso pelo programa abaixo?

```
1  #include <stdio.h>
2  struct Coordenada {
3      double x;
4      double y;
5  };
6  int main() {
7      struct Coordenada c1, c2, *c3;
8
9      c3 = &c1;
10     c1.x = -1;
11     c1.y = -1.5;
12
13     c2.x = 2.5;
14     c2.y = -5;
15     *c3 = c2;
16
17     printf("Coordenadas de c1: (%lf,%lf)\n", c1.x, c1.y);
18     return 0;
19 }
```

Para acessarmos os campos de uma variável **struct** via um ponteiro, podemos utilizar o operador ***** juntamente com o operador **.** como de costume:

```
1 Coordenada c1, *c3;  
2 c3 = &c1;  
3 (*c3).x = 1.5;  
4 (*c3).y = 1.5;
```

- Em C também podemos usar o operador `->`, que também é usado para acessar campos de uma estrutura via um ponteiro.

```
1 Coordenada c1, *c3;  
2 c3 = &c1;  
3 c3->x = 1.5;  
4 c3->y = 1.5;
```

- Resumindo: Para acessar campos de estruturas via ponteiros use um dos dois:
 - `ponteiroEstrutura->campo`
 - `(*ponteiroEstrutura).campo`

O que será impresso pelo programa abaixo?

```
1  ...
2  int main() {
3      struct Coordenada c1, c2, *c3, *c4;
4      c3 = &c1;
5      c4 = &c2;
6
7      c1.x = -1;
8      c1.y = -1.5;
9      c2.x = 2.5;
10     c2.y = -5;
11     (*c3).x = 1.5;
12     (*c3).y = 1.5;
13     c4->x = -1;
14     c4->y = -1;
15
16     printf("Coordenadas de c1: (%lf,%lf)\n",c1.x, c1.y);
17     printf("Coordenadas de c2: (%lf,%lf)\n",c2.x, c2.y);
18     return 0;
19 }
```

Podemos fazer alocação dinâmica de um vetor de registros da mesma forma que com tipos simples.

```
1 struct Aluno *vetAlu;  
2 vetAlu = malloc(10 * sizeof(struct Aluno));  
3 vetAlu[0].nota = 5.6;  
4 vetAlu[1].nota = 7.8;  
5 ...
```

Ponteiros e registros

Utilizando as funções criadas anteriormente podemos executar o exemplo:

```
1 struct Aluno leAluno();
2 void imprimeAluno(struct Aluno a);
3 void listarTurma(struct Aluno turma[], int n);
4
5 int main() {
6     struct Aluno *vetAlu;
7     int n, i;
8     printf("Numero de alunos: ");
9     scanf("%d", &n); getchar();
10
11     vetAlu = malloc(n * sizeof(struct Aluno)); /* Alocação dinâmica do
12     ↪ vetor de registros */
13     for(i = 0; i < n; i++)
14         vetAlu[i] = leAluno();
15     listarTurma(vetAlu, n);
16     free(vetAlu); /* Liberação de memória alocada */
17     return 0;
18 }
```

Exercícios

Exercício 1

- Crie um novo tipo de registro para armazenar coordenadas no plano cartesiano.
- Crie uma função para imprimir um ponto do tipo criado.
- Crie uma função para cada uma destas operações: soma de dois pontos, subtração de dois pontos, multiplicação por um escalar.

Informações extras: redefinição de tipos

- Às vezes, por questão de organização, gostaríamos de criar um tipo próprio nosso, que faz exatamente a mesma coisa que um outro tipo já existente.
- Por exemplo, em um programa onde manipulamos médias de alunos, todas as variáveis que trabalhassem com nota tivessem o tipo **nota**, e não **double**.

- A forma de se fazer isso é utilizando o comando **typedef**, seguindo a sintaxe abaixo:

```
1 typedef tipo_já_existente tipo_novo;
```

- Usualmente, fazemos essa declaração fora da função `main()`, embora seja permitido fazer dentro da função também. Veja o exemplo:

Exemplo: cria tipo **nota**

```
1  #include <stdio.h>
2
3  typedef double nota;
4
5  int main() {
6      nota p1;
7      printf("Digite a nota:");
8      scanf("%lf", &p1);
9      printf("A nota digitada foi: %lf\n", p1);
10     return 0;
11 }
```

Informações extras: exemplo de uso do typedef

- O uso mais comum para o comando **typedef** é para a redefinição de tipos registro.
- No nosso exemplo de **struct Aluno**, poderíamos redefinir este tipo para algo mais simples como simplesmente **Aluno**:

```
1  struct Aluno {
2      int ra;
3      double nota;
4  };
5
6  /* redefinimos tipo struct Aluno como Aluno*/
7  typedef struct Aluno Aluno;
8
9  int main () {
10     Aluno turma[10];
11     int i;
12     double media;
13     ...
14 }
```

Informações extras: exemplo de uso do typedef

```
1  #include <stdio.h>
2  struct Aluno {
3      int ra;
4      double nota;
5  };
6  typedef struct Aluno Aluno; /* redefine tipo struct Aluno como Aluno */
7  int main () {
8      Aluno turma[10];
9      int i;
10     double media = 0.0;
11     for (i = 0; i < 10; i++) {
12         scanf("%d", &turma[i].ra);
13         scanf("%lf", &turma[i].nota);
14     }
15     /* calcula a media da turma */
16     for (i = 0; i < 10; i++)
17         media = media + turma[i].nota;
18     media = media / 10.0;
19     printf("\nA média da turma é: %lf\n", media);
20     return 0;
21 }
```

Informações extras: tipos enumerados

- Para criar uma variável para armazenar um determinado mês de um ano (de janeiro a dezembro), uma das soluções possíveis é criar um inteiro e armazenar um número associado àquele mês. Assim, janeiro seria o mês número 1, fevereiro o mês número 2, e assim sucessivamente.
- Mas, o código seria mais claro se pudéssemos escrever algo como:

```
mes = janeiro;
```

Tipos enumerados: (enum)

- O comando `enum` cria um tipo enumerado: podemos usar nomes/identificadores para um conjunto finito de valores inteiros.
- Sua sintaxe é:

```
enum nomeDoTipo {identificador1, identificador2, ...  
    identificadorN};
```

- Exemplo:

```
1  /* criamos um novo tipo chamado "enum meses" */  
2  enum meses {jan, fev, mar, abr, mai, jun,  
3           jul, ago, set, out, nov, dez};
```

- O compilador associa o número 0 para o primeiro identificador, 1 para o segundo, etc.
- Variáveis do novo tipo criado são na realidade variáveis inteiras.
- Tipos enumerados são usados para deixar o código mais legível.

Tipos enumerados: enum

```
1  #include <stdio.h>
2  /* aqui criamos um novo tipo enumerado que pode ser usado por qualquer
3  função */
4  enum meses {jan, fev, mar, abr, mai, jun, jul, ago, set, out, nov, dec};
5
6  int main() {
7      enum meses a, b; /* cria 2 variáveis do tipo "enum meses" */
8
9      a = jan;
10     b = jun;
11
12     if (a != b) {
13         /* será impresso "0 é um mes diferente de 5" */
14         printf("%d é um mes diferente de %d\n", a, b);
15     }
16     return 0;
17 }
```

Usando um tipo enumerado

- Note que o primeiro identificador recebeu o valor zero, e demais identificadores receberam valores em sequência.
- Podemos alterar o valor inicial dos identificadores.

```
1  #include <stdio.h>
2  enum meses {jan = 1, fev, mar, abr, mai, jun, jul, ago, set, out, nov,
   ↪  dec};
3
4  int main() {
5      enum meses a, b; /* cria 2 variáveis do tipo "enum meses" */
6
7      a = jan;
8      b = jun;
9
10     if (a != b) {
11         /* será impresso "1 é um mes diferente de 6" */
12         printf("%d é um mes diferente de %d", a, b);
13     }
14     return 0;
15 }
```

- Um tipo enumerado pode ser criado para deixar o código mais legível.
- Variáveis de um tipo enumerado criado, são na realidade variáveis inteiras, mas temos a versatilidade de atribuir os identificadores do tipo enumerado para tais variáveis.