Programação Estruturada

Registros (*structs*)

Professores Emílio Francesquini e Carla Negri Lintzmayer

2018.Q3

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Registros

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:

```
struct nome_registro {
    tipo_1 nome_campo_1;
    tipo_2 nome_campo_2;
    tipo_3 nome_campo_3;
    ...
    tipo_n nome_campo_n;
};
```

- 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.

```
#include <stdio.h>

/* Declare tipos registro aqui */

int main() {
    /* Comandos */
}
```

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:

```c
#include <stdio.h>

struct Aluno {
    char nome[80];
    float nota;
};

int main() {
    /* variáveis são do tipo "struct Aluno" */
    struct Aluno a, b;
    ...
}
```

Utilizando os campos de um registro

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

```
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

```c
struct Aluno {
    char nome[45];
    float nota;
};

int main() {
    /* variáveis do tipo "struct Aluno" */
    struct Aluno a, b;
    a.nota = 4.7;
    b.nota = 2 * a.nota;
    return 0;
}
```

Utilizando os campos de um registro

```c
#include <stdio.h>
#include <string.h>
struct Aluno {
  char nome[45];
  float nota;
};
int main() {
  struct Aluno a, b;

  strcpy(a.nome, "Helen");
  a.nota = 8.6;
  strcpy(b.nome, "Dilbert");
  b.nota = 8.2;
  printf("a.nome = %s, a.nota = %f\n", a.nome, a.nota);
  printf("b.nome = %s, b.nota = %f\n", b.nome, b.nota);
  return 0;
}
```

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);
```

Atribuição de registros

- 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

```c
#include <stdio.h>
#include <string.h>

struct Aluno {
  char nome[80];
  float nota;
};
int main() {
  struct Aluno a, b;

  printf("Digite o nome:");
  fgets(a.nome, 80, stdin);
  printf("Digite a nota:");
  scanf("%f", &a.nota); getchar();

  b = a;  /* Atribuição de registros */

  printf("b.nome = %s, b.nota = %.2f\n", b.nome, b.nota);
}
```

Vetores e registros

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

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

  - Para usar:

```
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 e Funções

- 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.

Funções e registros

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:

```c
struct Aluno leAluno() {
    struct Aluno aluno;

    printf("Digite o Nome: ");
    fgets(aluno.nome, 80, stdin);
    printf("Digite a Nota: ");
    scanf("%f", &aluno.nota); getchar();

    return aux;
}
```

Funções e registros

```c
void imprimeAluno(struct Aluno a) {
    printf("Dados de um aluno --- ");
    printf("Nome: %s. Nota: %.2f\n", a.nome, a.nota);
}

void listarTurma(struct Aluno turma[], int n) {
    int i;
    printf("Imprimindo a turma\n");
    for (i = 0; i < n; i++)
        imprimeAluno(turma[i]);
}
```

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

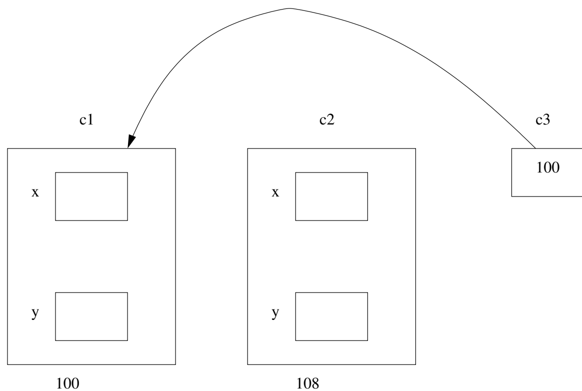
```
#include <stdio.h>
#include <string.h>
#define MAX 4
struct Aluno {
    char nome[80];
    float nota;
};
struct Aluno leAluno();
void imprimeAluno(struct Aluno a);
void listarTurma(struct Aluno turma[], int n);
int main() {
    int i;
    struct Aluno turma[MAX];
    for (i = 0; i < MAX; i++)
        turma[i] = leAluno();
    listarTurma(turma, MAX);
    return 0;
}
```

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



Ponteiros e Registros

O que será impresso pelo programa abaixo?

```c
#include <stdio.h>
struct Coordenada {
    double x;
    double y;
};
int main() {
    struct Coordenada c1, c2, *c3;

    c3 = &c1;
    c1.x = -1;
    c1.y = -1.5;

    c2.x = 2.5;
    c2.y = -5;
    *c3 = c2;

    printf("Coordenadas de c1: (%lf,%lf)\n", c1.x, c1.y);
    return 0;
}
```

Ponteiros e registros

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;
```

Ponteiros e registros

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

```
Coordenada c1, *c3;
c3 = &c1;
c3->x = 1.5;
c3->y = 1.5;
```

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

Ponteiros e registros

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 }
```

Ponteiros e registros

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

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

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
    ↪    vetor de registros */
12      for(i = 0; i < n; i++)
13          vetAlu[i] = leAluno();
14      listarTurma(vetAlu, n);
15      free(vetAlu);  /* Liberação de memória alocada */
16      return 0;
17  }
```

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

Informações extras: redefinido um tipo

- 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**.

Informações extras: o comando `typedef`

- 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:

Informações extras: o comando typedef

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

```c
#include <stdio.h>
struct Aluno {
    int ra;
    double nota;
};
typedef struct Aluno Aluno; /* redefine tipo struct Aluno como Aluno */
int main () {
    Aluno turma[10];
    int i;
    double media = 0.0;
    for (i = 0; i < 10; i++) {
        scanf("%d", &turma[i].ra);
        scanf("%lf", &turma[i].nota);
    }
    /* calcula a media da turma */
    for (i = 0; i < 10; i++)
        media = media + turma[i].nota;
    media = media / 10.0;
    printf("\nA média da turma é: %lf\n", media);
    return 0;
}
```

Informações extras: tipos enumerados

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};
```

Tipos enumerados: `enum`

- 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.

```c
#include <stdio.h>
enum meses {jan = 1, fev, mar, abr, mai, jun, jul, ago, set, out, nov,
 ↪  dec};

int main() {
    enum meses a, b; /* cria 2 variáveis do tipo "enum meses" */

    a = jan;
    b = jun;

    if (a != b) {
        /* será impresso "1 é um mes diferente de 6" */
        printf("%d é um mes diferente de %d", a, b);
    }
    return 0;
}
```

Tipos enumerados: resumindo

- 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.