COS110 – Algoritmos e Programação – Módulo 9

Priscila Machado Vieira Lima

Junho/2018

Curso : Engenharia de Controle e Automação

NCE

Universidade Federal do Rio de Janeiro

Roteiro

- **Estruturas**
- Campos de Bits
- Enumerações
- Uniões

Estruturas de dados

- Uma estrutura (struct) é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, agrupadas sob um único nome.

- Estruturas constituem um recurso importante para organizar os dados utilizados por um programa pois trata um grupo de valores como uma única variável.

- São chamadas de registros em outras linguagens de programação.

Estruturas de dados

- Uma estrutura (struct) é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, agrupadas sob um único nome.

- Estruturas constituem um recurso importante para organizar os dados utilizados por um programa pois trata um grupo de valores como uma única variável.

- São chamadas de registros em outras linguagens de programação.

Estruturas de dados

- Uma estrutura (struct) é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, agrupadas sob um único nome.

- Estruturas constituem um recurso importante para organizar os dados utilizados por um programa pois trata um grupo de valores como uma única variável.

- São chamadas de registros em outras linguagens de programação.

Estruturas de dados

- Estruturas (ou registros) são classificados como variáveis compostas heterogêneas, pois podem agrupar variáveis de tipos diferentes.

- Em contraposição, temos os vetores e matrizes, classificados como variáveis compostas homogêneas, pois somente agrupam variáveis do mesmo tipo.

Estruturas de dados

- Exemplo:

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

Estruturas de dados

- A palavra-chave **`struct`** informa ao compilador que um modelo de estrutura está sendo definido.

- "data" é uma etiqueta que dá nome à definição da estrutura.

- Uma definição de estrutura deve terminar em ponto-e-vírgula.

Estruturas de dados

- Os nomes declarados entre as chaves são os campos (ou membros) da estrutura.

- Os campos de uma mesma estrutura devem ter nomes diferentes.

- Porém, estruturas diferentes podem conter campos com o mesmo nome.

Estruturas de dados

- A definição de uma estrutura não reserva qualquer espaço na memória.

- Note que, no exemplo dado, nenhuma variável foi declarada de fato, apenas a forma dos dados foi definida.

- Essa definição, porém, cria um novo tipo de dados, que pode ser usado para declarar variáveis.

Declarando uma estrutura

- Duas maneiras de declarar a variável x do tipo data:

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

Dois comandos:
- Define estrutura como novo tipo
- Declara variável do novo tipo definido

ou

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

Um comando:
- Define estrutura e declara variável do novo tipo definido

Declarando uma estrutura

- Os campos de uma estrutura podem ser de qualquer tipo, inclusive uma estrutura previamente definida.

- Porém, o tipo dos campos não podem ser o do próprio tipo que está sendo definido.

Declarando uma estrutura

- A definição do formato de uma estrutura pode ser feita dentro da função principal (**`main`**) ou fora dela.

- Usualmente, declara-se fora da função principal, de modo que outras funções também possam "enxergar" a estrutura definida.

Estruturas de dados

- Forma geral de definição de uma estrutura:

```
struct <etiqueta> {
    <tipo> campo_1;
    <tipo> campo_2;
    ...
    <tipo> campo_n;
} <variáveis>;
```

Declarando uma estrutura utilizando **typedef**

- Novos tipos de dados podem ser definidos utilizando-se a palavra-chave **typedef**.

```
typedef struct nome_da_estrutura
{
    <tipo> campo_1;
    <tipo> campo_2;
    ...
    <tipo> campo_n;
} nome_do_tipo;
```

Declarando uma estrutura utilizando **typedef**

- Exemplo:

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

Declarando uma estrutura utilizando **typedef**

- O uso mais comum de **typedef** é com estruturas de dados, pois evita que a palavra-chave **struct** tenha de ser colocada toda vez que uma estrutura é declarada.

```
struct data dia_de_hoje;

tipoData dia_de_hoje;
```

Acessando os campos de uma estrutura

- Podemos acessar individualmente os campos de uma determinada estrutura como se fossem variáveis comuns.
- A sintaxe para acessar e manipular campos de estruturas é a seguinte:

```
<nome_da_variável>.<campo>
```

Leitura dos campos de uma estrutura

- A leitura dos campos de uma estrutura a partir do teclado deve ser feita campo a campo, como se fosse variáveis independentes.

```
printf ("Digite o nome do aluno: ");
scanf ("%s", &aluno.nome);

printf ("Digite a idade do aluno: ");
scanf ("%d", &aluno.idade);
```

Estruturas aninhadas

- Um campo de uma estrutura pode ser uma outra estrutura.
- Quando isso ocorre, temos uma estrutura aninhada.
- O padrão ANSI C especifica que as estruturas podem ser aninhadas até 15 níveis, mas a maioria dos compiladores permite mais.

Ponteiros para estruturas

- Como outros tipos de dados, ponteiros para estruturas são declarados colocando-se o operador * na frente do nome da variável estrutura:

```
struct data *ap_ontem;

tipoData *ap_amanha;
```

Ponteiros para estruturas

- Para acessar uma estrutura com ponteiros podem-se usar duas sintaxes:
  - Operador ponto:

```
*<ponteiro_estrutura>.<campo>
```

  - Operador seta:

```
<ponteiro_estrutura>-><campo>
```

Vetor de estruturas

- Usado quando precisamos de diversas cópias de uma estrutura.
- Por exemplo, cada cliente de uma locadora de vídeo constitui um elemento de um vetor, cujo tipo é uma estrutura de dados que define as características de cada cliente.

```
struct etiqueta variável[dimensão];
```

Passando estruturas para funções

- Podemos passar estruturas inteiras ou apenas campos destas como argumento para funções ou procedimentos.

- Tal passagem pode ser realizada por valor ou por referência.

- As estruturas manipuladas por mais de uma função devem ser declaradas globalmente, antes da definição da função.

Passando estruturas para funções

:: Passando campos por valor

- Este processo é realizado da mesma forma que para variáveis simples, atentando-se para as peculiaridades das estruturas de dados.

```
// Chamada
funcao(estrutura.campo, ...);

// Definicao formal
<tipo> funcao(<tipo> variavel, ...)
```

Passando estruturas para funções
:: Passando campos por referência

- Da mesma forma que acontece para variáveis simples, deve-se pôr o operador & antes do nome da estrutura na chamada da função.

```
// Chamada
funcao(&estrutura.campo, ...);

// Definicao formal
<tipo> funcao(<tipo> *variavel, ...)
```

Passando estruturas para funções

:: Por valor

- Ao contrário de arrays, estruturas podem ser passadas por valor como argumentos de função.
- Para estruturas grandes, há o problema da cópia de valores na pilha de memória.

```
// Chamada
funcao(estrutura, ...);

// Definicao formal
<tipo> funcao(struct <etiqueta> estrutura, ...)
```

Passando estruturas para funções
:: Por referência

- Permite alteração da estrutura sem necessidade de criação de uma cópia na memória.

```
// Chamada
funcao(&estrutura, ...);

// Definicao formal
<tipo> funcao(struct <etiqueta> *estrutura, ...)
```

Passando estruturas para funções :: Passando vetor de estruturas

- Funciona de forma semelhante a vetor de variáveis simples.
- Permite apenas passagem de parâmetros por referência, pois trata-se, afinal, de um vetor.

```
// Chamada
funcao(vet_de_struct, ...);

// Definicao formal
<tipo> funcao(struct <etiqueta> vetor_de_strut[], ...)
```

Passando estruturas para funções

- O tipo das estruturas reais (chamadas) e das formais (definição) deve ser o mesmo, ainda que sejam semelhantes.
- ????????? O QUE ISSO QUER DIZER ??????
- (mesmo que semanticamente as structs sejam compostas pelos mesmos sub-termos, na mesma ordem, não se pode considerá-las as mesmas)

Roteiro

- Estruturas
- **Campos de Bits**
- Uniões
- Enumerações

Campos de bits

- A Linguagem C possui um método específico para acessar um único bit dentro de um byte.
- Tal método é baseado em estruturas.
- Um campo de bits é um tipo de elemento da estrutura que define o comprimento do campo em bits.
- São frequentemente utilizados para manipular entradas de dispositivos de hardware (portas seriais e paralelas).

Campos de bits

- Um campo de bit deve ser declarado dentro da estrutura como **`unsigned`**, **`signed`** ou **`int`**.
- Se o comprimento do campo for de 1 bit, só pode ser declarado como **`unsigned`**.

```
struct status_type {
    int           campo1: 2;
    signed        campo2: 1;
    unsigned  campo3: 6;
};
```

Campos de bits

- É válido misturar elementos normais de estruturas com elementos de campos de bits.

```
struct empregado {
  struct   addr address;
  float    salario;
  unsigned lay_off  : 1; // ativo ou inativo
  unsigned hourly   : 1; // pago por hora ou mes
  unsigned deducoes : 3; // deducoes de impostos
};
```

Campos de bits

:: Restrições

- Não se pode obter endereço de uma variável de campos de bits.
- Não podem ser organizadas em matrizes.
- A soma dos comprimentos de todos os campos não pode ultrapassar o tamanho de um inteiro.
- A disposição dos bits (esquerda para direita ou direita para esquerda) varia de máquina para máquina.

Roteiro

- Estruturas
- Campos de Bits
- **Uniões**
- Enumerações

Union

- Uma **union** é uma posição de memória que é compartilhada por duas ou mais variáveis diferentes, geralmente de tipos diferentes, em momentos diferentes.
- Sua definição é semelhante à de estrutura:

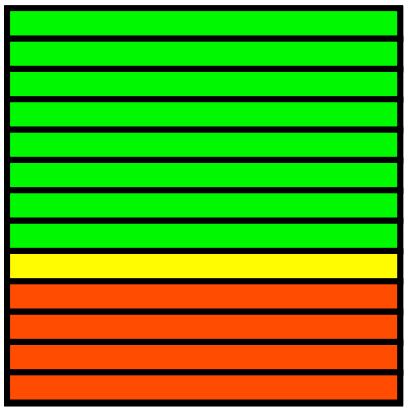
```
union <identificador> {
    <tipo> campo_1;
    <tipo> campo_2;
    ...
    <tipo> campo_n;
} <variáveis>;
```

Union

:: Exemplo

- O espaço ocupado por uma union na memória corresponde ao maior tamanho de variável que ela contém. Exemplo:

```
struct exemplo {
            int    a;
            char       b;
            double c;
}
```



```
union exemplo {
            int    a;
            char       b;
            double c;
}
```

Union

:: Exemplo

- Código para guardar um float e um inteiro em uma mesma posição de memória:

```
union ieee754 {
          float  num_real;
          int    num_hexa;
};
```

Roteiro

- Estruturas
- Campos de Bits
- Uniões
- **Enumerações**

Palavra chave **`enum`**

- **`enum`** é a abreviação de *ENUMERATE*.
- Serve para declarar e inicializar uma seqüência de constates inteiras.
- Principal vantagem – quando não se quer inicializar todas as constantes e cada uma precisa ter um valor único.
- Por padrão, a primeira constante equivale a zero. As restantes equivalem à anterior incrementada de um.

Palavra chave **enum**
:: Exemplo

```
enum colors {RED, GREEN, BLUE};
```

- **colors** é o nome dado para o grupo de constantes (opcional).
- Se não é atribuído um valor para as constantes, o valor padrão atribuído para o primeiro elemento na lista (RED em nosso caso) será 0 (zero).
- As demais constantes com valor indefinido terão valor o da constante anterior mais 1 (um).
- No nosso caso, GREEN = 1 e BLUE = 2.

Palavra chave **enum**

- Podem-se atribuir valores para constantes.

```
enum colors {RED = 1,
             YELLOW,
             GREEN = 6,
             BLUE};
```

- RED = 1, YELLOW = 2, GREEN = 6 e BLUE = 7.
- Geralmente, letras maiúsculas para dar nome a constantes.

Variáveis **enum**

- Constantes enumeradas são do tipo inteiro, então **`int x = RED;`** está correto.
- Porém, você pode criar o seu próprio tipo de dado, por exemplo **`colors`**.

```
enum colors corfundo;
```

- Declara uma variável chamada **`corfundo`**, a qual é do tipo de dado enumerado **`colors`**.