Introdução à Computação-II

Estruturas

Neste tópico abordaremos a manipulação estruturas em C.

Prof. Ciro Cirne Trindade

UNISANTOS

Estruturas

- **Estrutura** é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, colocadas juntas sobre um único nome
- As variáveis que fazem parte de uma estrutura são chamadas de **membros** da estrutura
- Exemplo: registro da folha de pagamento de um funcionário
  - nome (*string*)
  - número do departamento (inteiro)
  - salário (float)

Definindo um tipo estrutura

Especificação do tipo estrutura

Etiqueta da estrutura

```
struct funcionario {
    char nome[60];
    int departamento;
    float salario;
};
```

Membros da estrutura

Declarando uma variável do tipo estrutura

- A definição da estrutura não declara nenhuma variável
- Para declarar uma variável do tipo **`struct funcionario`**, fazemos:

```
struct funcionario func;
```

- ou

```
struct funcionario {
    /* definição dos membros */
} func;
```

Acessando Membros de uma Estrutura

- Para acessar um membro individual de uma estrutura usamos o operador ponto (.)
- Sintaxe:

variável-estrutura.nome-do-membro

- Exemplos:

```
func.departamento = 2;

printf("%.2f", func.salario);

fgets(func.nome, 60, stdin);
```

Inicializando Estruturas (1/2)

- A inicialização de estruturas é semelhante à inicialização de um vetor
- Exemplo:

```
struct data {
    int dia;
    char mes[10];
    int ano;
};

struct data natal = { 25, "Dezembro", 2018 };
struct data nasc = { 14, "Março", 2000 };
```

Inicializando estruturas (2/2)

- Também é possível inicializar os membros de uma estrutura fazendo referência ao nome desses membros
- Exemplo:

```
struct data natal = { .ano = 2018, .mes =
  "Dezembro"};
struct data nasc = { .mes = "Março", 2000 };
```

Atribuição entre Estruturas

- A informação contida em uma estrutura pode ser atribuída a outra estrutura do mesmo tipo usando uma simples expressão de atribuição, ou seja, você não precisa atribuir o valor de cada membro separadamente

Exemplo de atribuição entre estruturas

```
#include <stdio.h>
int main(){
    struct {
        int a;
        int b;
    } x, y;
    x.a = 10;
    x.b = 20;
    y = x;  // atribui uma estrutura a outra
    printf("Conteudo de y: %d %d.\n", y.a, y.b);
    return 0;
}
```

Estrutura sem etiqueta (anônima)

Vetores de Estruturas (1/2)

- Para declarar um vetor de estruturas, você deve primeiro definir a estrutura, e depois declarar um vetor deste tipo
- Exemplo:

```
struct funcionario func_array[100];
```

  - Cada elemento do vetor é uma estrutura do tipo **funcionario**
  - O nome **func_array** não é o nome de uma estrutura e sim o nome de um vetor em que os elementos são estruturas

Vetores de Estruturas (2/2)

- Para acessar uma estrutura específica dentro do vetor **`func_array`**, use o índice no nome da variável do tipo vetor
- Exemplo:

```
puts(func_array[2].nome);
```

Estruturas Encaixadas

- Podemos ter estruturas que contêm outras estruturas
- Exemplo:

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

struct circulo {
    struct ponto centro;
    float raio;
};
```

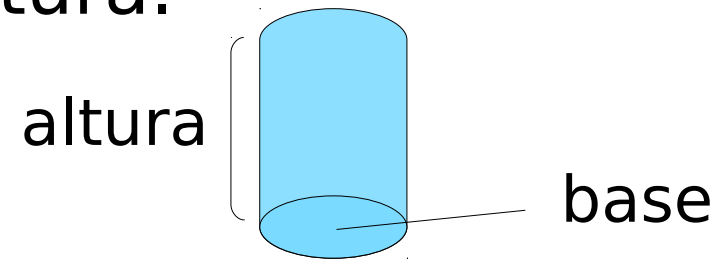
Exercícios (1/2)

1) Considere a estrutura `struct circulo` definida anteriormente e escreva um programa que dados dois círculos, verifique se eles se sobrepõem. Dois círculos se sobrepõem se a distância entre os centros destes círculos é menor que a soma de seus raios. Distância entre dois pontos:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Exercícios (2/2)

2) Defina uma estrutura para representar um cilindro. Um cilindro possui um círculo como base e uma altura.



- Escreva um programa que dadas as informações da base do cilindro e sua altura, determina o volume do cilindro.
  - Área da base do cilindro (A_b): πR^2
  - Volume do cilindro: A_b * altura

Ponteiro para estruturas

- Para declarar um ponteiro para a estrutura ponto fazemos:
  - `struct ponto * pponto;`
- Para acessar os membros da estrutura `struct ponto` através do ponteiro `pponto` fazemos:
  - `(*pponto).x = 12.0;`
  - `pponto->x = 12.0;`

Passando Estruturas para Funções

- Passando membros de estruturas para funções
  - Você está passando uma variável simples
- Passando estruturas inteiras para funções
  - Toda a estrutura é passada
- Passando estruturas por referência para uma função
  - Ponteiro para estruturas
- Devolvendo uma estrutura de uma função
  - A função deve ser do tipo estrutura

typedef (1/2)

- A palavra-chave **typedef** fornece um mecanismo para a criação de sinônimos
  - Os nomes dos tipos de estruturas são definidos frequentemente com **typedef** para criar nomes mais curtos de tipos
  - Por exemplo a instrução

```
typedef struct funcionario tfunc;
```

  - define o novo nome de tipo **tfunc** como um sinônimo do tipo **struct funcionario**

typedef (2/2)

- Os programadores C usam frequentemente **typedef** para definir um tipo estrutura de modo que não é exigido uma etiqueta da estrutura
- Por exemplo:

```
typedef struct {
    char nome[60];
    int departamento;
    float salario;
} tfunc;
```

Assim podemos fazer:
tfunc func;

Exercícios (1/2)

1) Considere a estrutura `ponto` para representar um ponto no espaço 2D e uma estrutura `retangulo` que representa um retângulo no espaço 2D. A estrutura `retangulo` possui 2 membros do tipo `ponto` que representam as coordenadas dos vértices superior esquerdo e inferior direto de um retângulo. Implemente uma função que indique se um determinado ponto *p* está localizado dentro ou fora de um retângulo *r*. A função deve devolver verdadeiro caso o ponto esteja localizado dentro do retângulo, ou falso caso contrário. Essa função deve obedecer ao protótipo:

- **`bool dentroRet(retangulo r, ponto p);`**

Exercícios (2/2)

2) Considere as declarações a seguir para representar o cadastro de alunos de uma disciplina e implemente uma função que imprima o número de matrícula, o nome, a turma e a média de todos os alunos aprovados na disciplina.

```
typedef struct {
    char nome[60];
    int matricula;
    char turma;
    float provas[3];
} aluno;
```

Considere que o critério de aprovação é que a média aritmética das 3 provas seja maior ou igual a 7.
A função deve receber como parâmetros o número de alunos da turma e um vetor com os dados dos alunos.

Uniões (1/4)

- Uma união é um tipo que permite armazenar diferentes tipos de dados na mesma área de memória (mas não simultaneamente)
- A definição de uma união é semelhante a de uma estrutura, substituindo a palavra-chave `struct` por **`union`**
- Exemplo:

Etiqueta da união

```
union uniao {
   int digito;
   double real;
   char letra;
};
```

Membros da união: pode armazenar um `int`, ou um `double` ou um `char`

Uniões (2/4)

- A declaração de variáveis do tipo união também é semelhante a declaração de variáveis do tipo estrutura
- Exemplos:

```
union uniao fit;

union uniao save[10];

union uniao *pu;
```

Declara uma variável simples do tipo união, o compilador aloca espaço suficiente para armazenar o maior dos membros da união (`double`, 64 bits)

Declara um vetor de tamanho 10, cada elemento com 64 bits

Declara um ponteiro para a união `union uniao` que pode armazenar o endereço desta união (64 bits, num SO de 64 bits)

UNISANTOS

Uniões (3/4)

- É possível inicializar uma união, entretanto, uma união armazena apenas um valor
- Para inicializar uma união há 3 possibilidades:
  - Com outra união

```
union uniao val_a;
val_a.letra = 'R';
union uniao val_b = val_a;
```

  - O primeiro membro:

```
union uniao val_c = { 99 };
```

  - Um membro específico:

```
union uniao val_d = {.real = 118.2 };
```

Uniões (4/4)

- Usando uma união:

```
union uniao u;
u.digito = 23;
u.real = 2.0;
u.letra = 'h';
```

23 é armazenado em `u` (4 bytes utilizados)

23 é sobrescrito, 2.0 é armazenado em `u` (8 bytes utilizados)

2.0 é sobrescrito, 'h' é armazenado em `u` (1 byte utilizado)

- Também é possível usar o `typedef` para definir um tipo união

```
typedef union {
    ...
} uniao;
```

Assim podemos fazer:
`uniao u;`

Enumerações (1/4)

- Você pode usar um tipo enumeração (**`enum`**) para declarar nomes simbólicos para constantes inteiras
- Usando a palavra-chave **`enum`** você cria um novo “tipo” e especifica os valores que ele pode assumir
- O propósito de tipos enumeração é aumentar a legibilidade de um programa

Enumerações (2/4)

- Forma geral da definição de um tipo enumeração:

```
enum etiqueta_da_enumeração {
   CONST_1[=int], CONST_2[=int], …,
   CONST_N[=int] };
```

- Por exemplo:

Define um tipo enumeração chamado **`enum cores`** que pode assumir os valores: `VERMELHO` (0), `LARANJA` (1), `AMARELO` (2), `VERDE` (3), `AZUL` (4) ou `VIOLETA` (5)

```
enum cores { VERMELHO, LARANJA, AMARELO,
             VERDE, AZUL, VIOLETA };

enum cores cor;
```

Declara uma variável chamada **`cor`** do tipo **`enum cores`**

Enumerações (3/4)

- Um tipo enumeração pode ser usado em instruções como as seguintes:

```
cor = AMARELO;
if (cor == AZUL) { … }
for (cor = VERMELHO; cor <= VIOLETA;
     cor++) { … }
```

- Por *default*, os valores das constantes de uma enumeração são inteiros de 0 ao número de constantes – 1
- É possível associar valores diferentes às constantes:

```
enum niveis { BAIXO = 100, MEDIO = 500,
              ALTO = 2000 };
```

Enumerações (4/4)

- Se você atribuir um valor a uma constante, mas não às constantes seguintes, as constantes seguintes serão enumeradas sequencialmente

```
enum felino { GATO, LEOPARDO = 10, PUMA,
              TIGRE };
```

- Então `GATO` é 0, por *default*, e `LEOPARDO`, `PUMA` e `TIGRE` são 10, 11 e 12, respectivamente

UNISANTOS

Referências

- CELES, W.; CERQUEIRA, R.; RANGEL, J.L.. “Introdução a Estruturas de Dados”. Elsevier, 2004.
- DEITEL, H.M.; DEITEL, P.J.. “Como Programar em C”. 2. ed., LTC, 1999.
- SHILDT, H.. “C Completo e Total”. 3. ed., Makron Books, 1996.
- PRATA, Stephen. “C Primer Plus”. 6. ed., Addison Wesley, 2014.