Algoritmo e Estrutura de Dados

Vagner Sacramento

Material preparado por:

Márcia Marra
DCC UFG

Estrutura de Dados

- Estruturas de dados e algoritmos estão intimamente ligados:
  - não se pode estudar estruturas de dados sem considerar os algoritmos associados a elas,
  - assim como a escolha dos algoritmos em geral depende da representação e da estrutura dos dados.
- Para resolver um problema é necessário escolher uma abstração da realidade, em geral mediante a definição de um conjunto de dados que representa a situação real.
- A seguir, deve ser escolhida a forma de representar esses dados.

Escolha da representação dos dados

- A escolha da representação dos dados é determinada, entre outras, pelas operações a serem realizadas sobre os dados.
- Considere a operação de adição:
  - Para pequenos números, uma boa representação é por meio de barras verticais (caso em que a operação de adição é bastante simples).
  - Já a representação por dígitos decimais requer regras relativamente complicadas, as quais devem ser memorizadas.
  - Entretanto, quando consideramos a adição de grandes números é mais fácil a representação por dígitos decimais (devido ao princípio baseado no peso relativo da posição de cada dígito).

Tipos de Dados

- Existem dois tipos principais de dados:
  - Tipos simples
  - Tipos estruturados
    - Em geral definem uma coleção de valores simples, ou um agregado de valores de tipos diferentes.

Tipos Abstratos de Dados (TAD's)

- Modelo matemático, acompanhado das operações definidas sobre o modelo.
- Exemplo:
  - o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação.
- TAD's são utilizados extensivamente como base para o projeto de algoritmos.
- A implementação do algoritmo em uma linguagem de programação específica exige a representação do TAD em termos dos tipos de dados e dos operadores suportados.

Tipos Abstratos de Dados (TAD's)

- A representação do modelo matemático por trás do tipo abstrato de dados é realizada mediante uma estrutura de dados.
- Podemos considerar TAD's como generalizações de tipos primitivos e procedimentos como generalizações de operações primitivas.
- O TAD encapsula tipos de dados. A definição do tipo e todas as operações ficam localizadas numa seção do programa.

Tipos Abstratos de Dados em C

- Em C, utilizamos o conceito de estrutura para definir tipos abstratos de dados (TAD's).
- Uma estrutura agrupa várias variáveis numa só.
- Funciona como uma ficha pessoal que tenha nome, telefone e endereço.
- A ficha seria uma estrutura. A estrutura, então, serve para agrupar um conjunto de dados não similares, formando um novo tipo de dados.

Estruturas (Structs)

- São variáveis compostas heterogêneas
- Visa facilitar manipulação de um conjunto de dados logicamente relacionados, mas de diferentes tipos.
- Para se utilizar uma variável heterogênea é preciso:
  - Passo1: definir a estrutura
  - Passo2: declarar as variáveis que terão esta estrutura

Criando uma estrutura

- Para se criar uma estrutura usa-se o comando **struct**. Sua forma geral é:

```
struct <nome_estrutura> {
    <tipo_de_dados> <nome da variável>;
    <tipo_de_dados> <nome da variável>;
    <tipo_de_dados> <nome da variável>;
    . . .
} variaveis_estrutura;
```

- O nome_estrutura é o nome para a estrutura. As variáveis_estrutura são opcionais e seriam nomes de variáveis que o usuário já estaria declarando e que seriam do tipo nome_estrutura.

Exemplo de estruturas

```
struct est{
      int i;
      float f;
} a, b;

struct tipo_endereco {
  char rua [50];
  int numero;
  char bairro [20];
  char cidade [30];
  char sigla_estado [3];
  long int CEP;
};
```

Exemplos de estruturas

- Vamos agora criar uma estrutura chamada ficha_pessoal com os dados pessoais de uma pessoa:

```
struct ficha_pessoal {
  char nome [50];
  long int telefone;
  struct tipo_endereco endereco;
};
```

Utilizando as estruturas

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

struct tipo_endereco {
   char rua [50];
   int numero;
   char bairro [20];
   char cidade [30];
   char sigla_estado [3];
   long int CEP;
};
struct ficha_pessoal {
   char nome [50];
   long int telefone;
   struct tipo_endereco endereco;
};
```

Exemplo utilizando estruturas

```
int main () {
  struct ficha_pessoal ficha;
  strcpy (ficha.nome,"Luiz Osvaldo Silva");
  ficha.telefone=4921234;
  strcpy (ficha.endereco.rua,"Rua das
  Flores");
  ficha.endereco.numero=10;
  strcpy (ficha.endereco.bairro,"Cidade
  Velha");
  strcpy (ficha.endereco.cidade,"Belo
  Horizonte");
  strcpy (ficha.endereco.sigla_estado,"MG");
  ficha.endereco.CEP=31340230;
  return 0;
}
```

Acessando as estruturas

- O programa declara uma variável ficha do tipo **ficha_pessoal** e preenche os seus dados.
- O exemplo mostra como podemos acessar um elemento de uma estrutura: basta usar o ponto (**.**).
- Assim, para acessar o campo telefone de ficha, escrevemos:

```
ficha.telefone = 4921234;
```

Acessando as estruturas

- Como a struct ficha pessoal possui um campo, endereco, que também é uma struct, podemos fazer acesso aos campos desta struct interna da seguinte maneira:

ficha.endereco.numero = 10;

ficha.endereco.CEP=31340230;

- Desta forma, estamos acessando, primeiramente, o campo endereco da struct ficha e, dentro deste campo, estamos acessando o campo numero e o campo CEP.

Matrizes de Estruturas

- Um estrutura é como qualquer outro tipo de dado no C. Podemos, portanto, criar matrizes de estruturas.
- Vamos ver como ficaria a declaração de um vetor de 100 fichas pessoais:

```
struct ficha_pessoal fichas [100];
```

- Poderíamos então acessar a segunda letra da sigla de estado da décima terceira ficha fazendo:

```
fichas[12].endereco.sigla_estado[1];
```

Pratique o conceito de estruturas

- Escreva um programa fazendo o uso de struct's.
  - Você deverá criar uma struct chamada Ponto, contendo apenas a posição x e y (inteiros) do ponto.
  - Declare 2 pontos, leia a posição (coordenadas x e y) de cada um e calcule a distância entre eles.
  - Apresente no final a distância entre os dois pontos.

Atribuindo estruturas

- Podemos atribuir duas estruturas que sejam do *mesmo* tipo.
- O C irá, neste caso, copiar uma estrutura, campo por campo, na outra.

Exemplo

```
struct est1 {
   int i;
   float f;
};
void main()
{
   // Declara primeira e segunda como structs
   struct est1 primeira, segunda;
   primeira.i = 10;
   primeira.f = 3.1415;
   segunda = primeira;
   // A segunda struct e' agora igual a primeira
   printf(" Os valores armazenasdos na segunda struct
   sao :  %d  e  %f ", segunda.i , segunda.f);
}
```

Atribuindo estruturas

- Na atribuição de estruturas, todos os campos de primeira serão copiados na segunda.
- Note que **isto é diferente do que acontecia em vetores**, onde, para fazer a cópia dos elementos de um vetor em outro, tínhamos que copiar elemento por elemento do vetor.
- Nas structs é muito mais fácil!

Argumentos para funções

- Nos exemplos apresentados anteriormente, vimos comandos da seguinte forma:

```
strcpy (ficha.nome,"Luiz Osvaldo Silva");
```

- Neste comando um elemento de uma estrutura é passado para uma função. Este tipo de operação pode ser feita sem maiores considerações.
- Podemos também passar para uma função uma estrutura inteira.

Argumentos para funções

```
void PreencheFicha (struct ficha_pessoal ficha) {
    ...
    ...
}
```

- Como vemos acima é fácil passar a estrutura como um todo para a função.
- Devemos observar que, como em qualquer outra função no C, a passagem da estrutura é feita por valor.
- A estrutura que está sendo passada, vai ser copiada, campo por campo, em uma variável local da função PreencheFicha.
- Isto significa que alterações na estrutura dentro da função não terão efeito na variável fora da função.

Ponteiros e estruturas

- Podemos ter um ponteiro para uma estrutura.
- Como poderia ser declarado um ponteiro para as estruturas de ficha que estamos usando nestas seções?
- Da mesma forma como declaramos ponteiros para outras variáveis:

```
struct ficha_pessoal *p;
```

Ponteiros e estruturas

- Os ponteiros para uma estrutura funcionam como os ponteiros para qualquer outro tipo de dados no C.
- Para usá-lo, existe duas possibilidades:
  - A primeira é apontá-lo para uma variável struct já existente, da seguinte maneira:

```
struct ficha_pessoal ficha;
struct ficha_pessoal *p;
p  = &ficha;
```

Ponteiros e estruturas

- Há mais um detalhe a ser considerado.
- Se apontarmos o ponteiro **p** para uma estrutura qualquer (como fizemos em p = &ficha;) e quisermos acessar um elemento da estrutura poderíamos fazer:

```
(*p).nome
```

- Os parênteses são necessários, porque o operador . tem precedência maior que o operador * .

Ponteiros e estruturas

- Porém, este formato não é muito usado.
- O que é comum de se fazer é acessar o elemento **nome** através do operador seta, que é formado por um sinal de "menos" (-) seguido por um sinal de "maior que" (>), isto é: **->** .
- Assim faremos:

```
p->nome
```

- A declaração acima é muito mais fácil e concisa. Para acessarmos o elemento **CEP** dentro de **endereco** faríamos:

```
p->endereco.CEP
```

Comando typedef

- O comando **typedef** permite ao programador definir um novo nome para um determinado tipo.
- Sua forma geral é:

```
typedef antigo_nome novo_nome;
```

- Como exemplo vamos dar o nome de **inteiro** para o tipo **int**:

```
typedef int inteiro;
```

- Agora é possível declarar o tipo **inteiro**.

Comando typedef e estruturas

- O comando **typedef** também pode ser utilizado para dar nome a tipos complexos, como as estruturas.
- As estruturas criadas anteriormente poderiam ser definidas como tipos através do comando **typedef**.

Exemplo

```
#include <stdio.h>
typedef struct tipo_endereco
{
   char rua [50];
   int numero;
   char bairro [20];
   char cidade [30];
   char sigla_estado [3];
   long int CEP;
} TEndereco;
typedef struct ficha_pessoal
{
   char nome [50];
   long int telefone;
   TEndereco endereco;
} TFicha;
void main(void)
{
   TFicha *ex;
   ...
}
```