Capítulo 10

Outros Tipos de Dados

" Testes experimentais de programas podem, quando muito mostrar a presença de erros, mas nunca podem mostrar a sua ausência "

- Niklaus Wirth -

Sumário:

10.1- Estruturas

Nas linguagens de programação, podemos definir um tipo de dados, cujos campos são valores de tipos de dados diferentes. Esse tipo de dados é denominado por registro, mas para a linguagem C é denominado por estruturas e possui a seguinte sintaxe:

```
struct TipoDoRegistro
{
  tipo1 nomeDoCampo1;
  tipo2 nomeDoCampo2;
    :
  tipon nomeDoCampon;
} variável;
```

Para consolidar esse conceito, vamos considerar o desenvolvimento de um programa para manipular datas. Uma data é composta por três campos do tipo inteiro, o dia, o mês e o ano. Se não houvesse um mecanismo para associar essas variáveis teríamos de declarar as variáveis:

```
int dia, mes, ano.
```

Como essas variáveis não têm qualquer relação entre si, se tivéssemos de desenvolver um programa para calcular a idade de uma pessoa, teríamos de declarar seis variáveis, e o programador, teria a responsabilidade de não misturar o conteúdo das variáveis que fazem referencia a data actual, com as variáveis que fazem referência a data de nascimento.

Para tornar o código mais legível e facilitar a vida do programador, a linguagem C oferece um recurso que permite associar essas variáveis a uma única entidade. Com esse recurso, essa variável pode ser declarada da seguinte forma:

```
struct TData
{
   int dia, mes, ano;
};
```

Com essa instrução, informamos ao compilador que estamos em presença de uma estrutura (registro). O compilador calcula a quantidade de memória necessária para sua manipulação, mas não reserva esse espaço em memória.

Para reservar o espaço de memória necessário para sua manipulação, é necessário declarar as variáveis associadas ao tipo de dados TData. Essa declaração é feita com a seguinte instrução:

```
struct TData   DtNasc, DtActual;
```

As variáveis do tipo estrutura, podem ser utilizadas para operações de atribuição, leitura, impressão, etc. Mas para efectuarmos essas operações, devemos utilizar a seguinte notação:

```
DtNasc.dia  = 22;
DtNasc.mes = 12;
DtNasc.ano  = 1990;
```

Com essas instruções, armazenamos no registro DtNasc, no campo dia o valor 22, no campo mês o valor 12, e no campo ano o valor 1990. Enquanto na instrução:

```
DtActual = DtNasc;
```

Copiamos todos os campos da estrutura DtNasc para a estrutura DtActual; as linguagens de programação em geral, permitem que uma estrutura possa ser atribuído a outra variável estrutura, se essas estruturas forem do mesmo tipo.

Vamos estudar em seguida, um exemplo que mostra o conceito de estruturas aninhadas, ou seja estruturas dentro de estruturas. Por definição, uma estrutura pode conter variáveis do tipo elementar, vectores, ponteiros e outras estruturas.

Mas, para incluirmos uma variável do tipo estrutura no interior de uma outra variável do mesmo tipo, a variável que será incluída deve ser declarada em primeiro lugar. Por exemplo, dada a definição da estrutura:

```
struct TData
 {
   int dia , mes , ano;
 };
```

e a estrutura:

```
struct TEndereco
 {
   char rua[100];
    char bairro[100];
    char cidade[100];
    int cpostal;
 };
```

podemos utilizar essas definições para incluir na estrutura TAluno essas estruturas.

```
struct  TAluno
 {
    char nome[100];
    int numero;
    char sexo;
    int curso;
    struct TData nascimento;
    struct TEndereco endereco;
 };
```

Para carregar os dados de um hipotético estudante, basta executar o seguinte programa.

```
int main()
{
   struct TAluno  aluno;
   strcpy (aluno.nome,"Fulano de Tal");
   aluno.numero = 3209918;
   aluno.sexo = 'M';
   aluno.curso = 45070;
   aluno.nascimento.dia = 13;
   aluno.nascimento.mes = 9;
   aluno.nascimento.ano = 1980;
   strcpy (aluno.endereco.rua,"Av. da Liberdade “);
   strcpy (aluno.endereco.bairro,"Maculusso");
   strcpy (aluno.endereco.cidade,"Luanda");
   aluno.endereco.cpostal = 3742;
```

```
    ...
}
```

Contudo, a principal desvantagem da utilização de estruturas, está na declaração das variáveis que têm de ser precedidas pela palavra reservada **struct**.

Para declararmos uma variável do tipo estrutura com uma sintaxe igual as variáveis do tipo elementar, necessitamos de utilizar a palavra reservada **typedef**. Por exemplo:

```
typedef struct
{
   int    dia, mes, ano;
} TData;
```

Agora podemos associar a essa estrutura as variáveis DtNasc e DtActual, com a mesma sintaxe das variáveis do tipo elementar, ou seja:

```
TData DtNasc, DtActual;
```

10.2- Passagem de Parâmetros

A passagem de parâmetros para as variáveis do tipo estrutura é feita de forma similar a passagem de parâmetros para as variáveis do tipo elementar. Se a estrutura for passada por valor, o acesso ao conteúdo de qualquer campo é feito pelo nome da estrutura, seguido do operador ponto e o nome do campo, em termos simbólicos:

```
nomeDoRegistro.nomeDoCampo;
```

Mas, se a estrutura for passada por referência, o acesso ao conteúdo de qualquer campo é feito pelo nome da estrutura, seguido do operador seta e o nome do campo, em termos simbólicos:

```
nomeDoRegistro->nomeDoCampo;
```

10.3- Vectores de Estruturas

Antes de desenvolvermos algumas operações com vectores de estruturas, vamos estudar como declarar a estrutura de dados onde essas operações serão realizadas.

Utilizaremos para o efeito, vamos declarar um vector de estruturas com MAX elementos, onde cada elemento é uma estrutura com dois campos: chave do tipo inteiro, cujo conceito veremos mais tarde, e valor que não têm de momento qualquer importância de momento.

Associado a esse vector temos uma variável, denominada por nElem, que dar-nos-á em qualquer instante o número de elementos inseridos, e para terminar, vamos declarar uma variável do tipo enumerado (investigue esse assunto), que simula os valores lógicos da álgebra booleana.

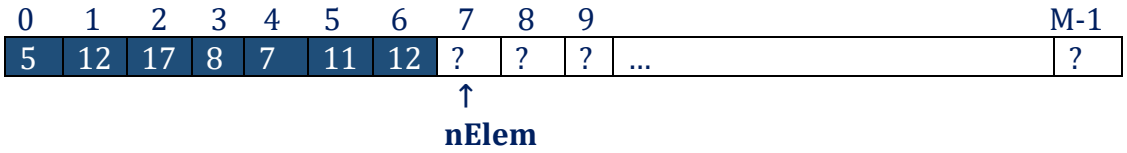
Declaramos em primeiro lugar, um elemento do vector.

```
typedef
{
   int chave;
   float valor;
 } TItem;
```

Em seguida, declaramos um vector que contêm esses elementos e relacionamos esse vector com uma variável que dar-nos-á o número de elementos inseridos.

```
typedef
{
   TItem item[MAX];
   int nElem;
} TVet;
```

Em termos gráficos



onde

```
#define MAX   1000
```

e

$0 \leq nElem \leq MAX$

Para terminar, vamos declarar a variável do tipo enumerado que simula os valores lógicos.

```
typedey enum {FALSE = 0, TRUE = 1} Boolean;
```

A passagem de parâmetros dos vectores de estruturas é feita de forma análoga a passagem de parâmetros das variáveis elementares.Por seja vet um vector do tipo **tVet**. Para mostrar na tela o conteúdo dos seus elementos, utilizamos a passagem por valor, e no cabeçalho da função esse vector será declarado como:

```
void mostrarDados ( TVet vet )
```

Mas, para inserir um conjunto de elementos nesse vector, utilizamos a passagem por referência, e o cabeçalho da função esse vector será declarado como:

```
void inserirDados ( TVet *vet )
```

10.4 - Operações Com Vectores não Ordenados

A **operação de inicialização**, consiste em preparar o vector de estruturas para receber dados. Como nos vectores tradicionais, a variável que determina o número de elementos inseridos é igual a zeros. Queremos com isso dizer que o conteúdo dos elementos que estão nas posições 0 até MAX-1, são considerados de lixo residual.

```
void inicializarVector (TVet  *vet)
 {
   vet->nElem  = 0;
 }
```

A **operação para determinar se um vector de estruturas está vazio**, consiste em verificar se ele não possui elementos. Pelo operador anterior, basta verificar se o número de elementos inseridos é igual a zero.

```
Boolean estaVazio (Tvet vet)
 {
   return ( vet.nElem  == 0 );
 }
```

A **operação para determinar se um vector de estruturas está cheio**, consiste em verificar se todos os seus elementos estão preenchidos. Para isso, basta verificar se o número de elementos inseridos é igual a dimensão do vector.

```
Boolean estaCheio (Tvet vet)
 {
   return ( vet.nElem  == MAX );
 }
```

A **operação para aceder à um determinado registro**, é muito simples. Ela consiste em retornar o conteúdo do elemento que pretendemos aceder, e que cuja localização foi passada como parâmetro.

Mas, antes de efectuar essa operação é necessário verificar se essa localização está contida no intervalo {0,…,nElem-1}. Se a condição for falsa emitir a correspondente mensagem de erro.

```
Tvet acederElemento (Tvet vet,  int pos)
 {
    if ( estaContido(0, vet.nElem, pos) )
       return vet.item[pos];
    printf ("\n Erro: posição invalida ");
 }
```

Antes de iniciarmos o estudo das próximas operações, vamos desenvolver uma função primitiva, do tipo booleano, denominada por estaContido() que recebe como parâmetro o início do intervalo (ini), o fim do intervalo (fim) e a posição de inserção (pos). Retorna verdadeiro se ini ≤ pos ≤ fim e falso no caso contrário.

```
boolean estaContido (int ini, int fim, int pos)
 {
   return ( pos >= ini && pos <= fim);
 }
```

A **operação para inserir um registro dado uma chave**, também é muito simples. Ela consiste em inserir o registro que contêm essa chave na primeira posição com lixo residual.

Então o nosso problema consiste em desenvolver um método para determinar a primeira posição com lixo residual. Pela declaração dessa estrutura a primeira posição com lixo residual está referenciada pela variável nElem. Mas essa variável só faz referencia a essa localização, se durante o processo de inserção, os elementos forem armazenados em posições consecutivas, e a variável actualizada em mais uma unidade.

```
void inserirDadoChave ( Tvet *vet , TItem x)
 {
    vet->item[vet->nElem] = x;
    vet->nElem++;
 }
```

A **operação para imprimir todos os elementos de um vector de estruturas**, consiste em percorrer o vector do primeiro até ao último elemento inserido, e mostrar na tela o conteúdo de cada registro.

```
void imprimirVector (TVet vet)
 {
     for (int i = 0; i < vet.nElem ; i++)
        printf(" %d  %f ", vet.item[i].chave, vet.item[i].valor);
 }
```

A **operação para inserir um elemento dada uma determinada posição**, é mais complexa. Ela consiste em deslocar para à direita, todos os elementos que vão desde o último de elemento inserido até a posição de inserção. Inserir em seguida o elemento na posição de inserção e actualizar índice que faz referência ao número de elementos inseridos.

Mas, antes de efectuar essa operação é necessário verificar se o vector não está cheio, e se a posição (índice) de inserção está contida no intervalo {0,…,nElem}. Se essas condições forem falsas, devemos emitir as correspondentes mensagens de erro.

```
void inserirNaPosicao (TItem x , int pos , TVet *vet )
{
   int i;
   if (estaCheio(vet))
      printf ("\n Erro: vector cheio");
    else if (estaContido (0, vet.nElem, pos) )
            {
              for ( i = vet->nElem-1; i >= pos ; i -- )
                  vet->item[i+1] = vet->item[i];
              vet->item[pos] = x;
              vet->nelem ++;
             }
          else
            printf (" \n Erro: Insercao não permitida- fora do intervalo ");
}
```

A **operação para remover um elemento dada uma determinada posição**, consiste em guardar a informação que se pretende remover numa variável auxiliar, deslocar em seguida, todos os elementos que vão desde a posição de remoção até a posição do último elemento inserido para a esquerda. Para terminar, actualizar o índice que faz referência ao número de elementos inseridos.

Mais antes de efectuar essa operação é necessário verificar se o vector não está vazio, e se o índice de remoção está contida no intervalo {0,..,nElem-1}. Se essas condições forem falsas, devemos emitir as correspondentes mensagens de erro.

```
void RemoverDaPosicao (int pos , TVet *vet , Titem *x)
 {
   int i;
   if ( estaVazio (vet) )
      printf ("\n Erro: Vector Vazio ");
   else if  ( estaContido (0, vet.nElem-1, pos))
            {
              *x = vet->item[pos];
              for ( i = pos ; i < vet->nElem-1 ; i++ )
                   vet->item[i] = vet->item[i+1];
              vet->nElem--;
             }
         else
            printf ("\n Erro: Remocao não permitida- Fora do intervalo");
  }
```

Devemos salientar que com uma única declaração, passamos como parâmetro um vector com MAX elementos e uma variável que faz referência ao número de elementos inseridos nesse vector. Essa técnica, denotada por herança, é uma das bases da programação orientada por objectos.

10.5 - Operações com Vectores Ordenados

Seja c_0, c_1, c_2, …., $c_{nElem-1}$, números inteiros diferentes, que são os conteúdos das chaves de um vector de estruturas. Dizemos que esse vector está ordenado em ordem crescente se:

$$c_0 \leq c_1 \leq c_2 \leq \ldots \leq c_{NElem-1}$$

As funções de inicialização(), estacheia(), estaVazia(), imprimirVector() são análogas as correspondentes funções para vectores de estruturas não ordenados. Mas as operações para inserir e remover um novo registro dado uma chave apresentam diferenças consideráveis.

A **operação para inserir um registro dado uma chave**, consiste em percorrer o vector até encontrar a posição de inserção.

A localização da posição de inserção depende do valor da chave que pretendemos inserir. Se a chave for maior do que todas as chaves inseridas, essa posição está localizada no elemento de índice nElem. Mas se a chave não for maior do que todas as chaves inseridas, então a posição de inserção, esta localizada no primeiro elemento cujo conteúdo da chave, é maior do que a chave que pretendemos inserir.

O processo de inserção propriamente dito consiste em deslocar todos os elementos que vão desde o número de elementos inseridos menos uma únidade até a posição de inserção, para a direita. Em seguida, inserir o elemento na posição de inserção e para finalizar, actualizar o índice que faz referência ao número de elementos inseridos

Mas, antes de efectuar essa operação é necessário verificar se o vector não está cheio.

```
void inserirElemento (TItem x , TVet *vet )
{
   int i ,  pos;
   if (estaCheio(vet))
      printf ("\n Erro: vector cheio");
    else
       {
          pos = 0;
          while ( pos < vet->nElem  && vet->item[pos]. chave < x.chave)
                    pos++ ;
           if  ( vet->item[pos].chave == x.chave)
               printf ("\n Erro: Chave já inserida ");
           else
               {
                 for ( i = vet->nElem-1; i >= pos ; i -- )
                      vet->item[i+1] = vet->item[i];
```

```
                vet->item[pos] = x;
                vet->nelem ++;
            }
        }
    }
```

A **operação para remover um elemento dada uma chave**, consiste em percorrer o vector até localizar a posição de remoção.

A posição de remoção depende do valor da chave que se pretende inserir. Se a chave for maior do que todas as chaves inseridas o percurso termina quando for encontrada o elemento cujo índice é igual a nElem. Se a chave não for maior do que as chaves já inseridas, temos duas situações. A chave não se encontra no vector, e nessa altura o percurso termina quando encontrarmos um elemento cuja chave é maior do que a chave que procuramos, ou a chave encontra-se no vector, e nessa altura o percurso termina quando encontrarmos um elemento cuja chave é igual a chave que procuramos.

O processo de remoção propriamente dito, consiste em armazenar o registro que contêm essa chave numa variável auxiliar; deslocar em seguida todos os elementos que vão desde a posição de remoção até a posição que antecede o número de elementos inseridos para a esquerda, e para terminar, actualizar o índice que faz referência ao número de elementos inseridos.

Mais antes de efectuar essa operação é necessário verificar se o vector não está vazio. Para consolidar a matéria deixamos essa operação como exercício.

10.6 - Recomendações Bibliográficas

Para o leitor aprofundar os seus conhecimentos sobre vectores e recursão, recomendamos os seguintes livros:

Celes W., Cerqueira R., Rangel J. L., - *Introdução a Estruturas de Dados com Técnicas de Programação em C*, 2ª Edição, Editora Elsevier, Brasil, 2016.

King K. N. ;- *C Programming – A Modern Approach*, W. W. Norton & Company, Inc., 2nd edition, 20010.

Feofiloff P.;- *Algoritmos em Linguagem C*, Editora Campos, Brasil, 2009.

10.7- Exercícios Propostos

10.7.1-Desenvolva uma função que recebe um vector de estruturas não ordenado e uma determinada posição. Devolva o conteúdo dessa posição.

10.7.2-Desenvolva uma função que recebe um vector de estruturas. Verifique se esse vector está ordenado em ordem decrescente.

10.7.3-Desenvolva uma função que recebe um vector de estruturas não ordenado e uma determinada chave. Inserir essa chave no vector se ela ainda não foi inserida.

10.7.4-Desenvolva uma função que recebe um vector de estruturas não ordenado e dois registros, r1 e r2. Inserir o registro r2 depois de ter encontrado o registro r1. Se o registro r1 não estiver contido no vector, emitir a correspondente mensagem de erro.

10.7.5-Desenvolva uma função que recebe um vector de estruturas ordenado em ordem decrescente e uma determinada chave. Remover essa chave. Estamos a supor que a chave que pretendemos remover pode não estar contida na vector.

10.7.6-Desenvolva uma função que recebe um vector de estruturas ordenado com chaves duplicadas. Apagar desse vector todas as chaves duplicadas.

10.7.7-Desenvolva uma função que recebe dois vectores de estruturas ordenados em ordem crescente. Devolver um terceiro vector também ordenado que é a intercalação dos dois vectores, mas esse vector não pode conter chaves duplicadas.

10.7.8-Desenvolva uma função que recebe um vector de estruturas e copia para um outro vector de estruturas todos os seus elementos, mas elimina as chaves repetidas.

10.7.9-Desenvolva uma função que recebe dois vectores de estruturas ordenados. Copiar para um terceiro vector os elementos comuns. Esse terceiro vector também deverá estar ordenado.

10.7.10-Desenvolva uma função que recebe um vector de estruturas não ordenado e um número inteiro n. Remova desse vector os n primeiros elementos e os insira no fim do vector.

10.7.11-Desenvolva uma função que recebe um vector de estruturas ordenado em ordem crescente. Alterar o conteúdo do campo chave de tal modo que o vector continue ordenado.

10.7.12-Desenvolva uma função que recebe um vector de estruturas e um determinado registro. Encontrar no vector um registro com a mesma chave e alterar o conteúdo do campo valor.

10.7.13-Desenvolva uma função que recebe um vector de estruturas. Reorganize esse vector de tal modo que os elementos antes do meio possuem uma chave inferior ao meio e os elementos depois do meio possuem uma chave superior ao mio.

10.7.14-Os sistemas operativos registram em um arquivo as atividades dos utilizadores. Por exemplo, num arquivo do tipo log, arquivo no formato texto, pode ter registrado os nomes de todos os utilizadores que efetuaram um login

(entrada) num período, a data do login, o horário do login e o horário do logout (saída). Este arquivo pode ser examinado em encontrar de atividades suspeitas. Desenvolva um programa para auxiliar o administrador nesta tarefa. Considere a existência de um arquivo fictício onde estão armazenados, por cada linha, o nome (login) do utilizador, a data de login, o horário de login e o horário do logout. Construir as estruturas de dados apropriadas para armazenar esses dados. Ler o arquivo e preencher um vector de estruturas. Implementar em seguida, as seguintes funções: mostraUtilizador() que tem por finalidade, mostra na tela os nomes de todos os utilizadores que estão armazenados no arquivo, desprezando os nomes repetidos; mostraAposHora() que tem por finalidade mostrar na tela os nomes de todos os utilizadores que fizeram um login após uma determinada hora indicada pelo administrador; mostraAcessoUtilizador() que tem por finalidade, mostrar na tela todos os acesso feitos por um determinado utilizador indicado pelo administrador.