Tipos Abstratos de Dados em C

Voltar para DAS5102 Fundamentos da Estrutura da Informação

Índice

Tipos Abstratos de Dados - TAD

É uma especificação de um conjunto de dados e operações que podem ser executadas sobre esses dados. Além disso, é uma metodologia de programação que tem como proposta reduzir a informação necessária para a criação/programação de um algoritmo através da abstração das variáveis envolvidas em uma única entidade fechada com operações próprias à sua natureza.

A idéia que permeia os tipos abstratos de dados é a de que os tipos simples ou atômicos suportados diretamente pela maioria das linguagens de programação nem sempre são suficientemente expressivos para representar e manipular tipos de dados mais complexos recorrentemente requeridos por programas de computador. Desta forma, novos tipos de dados devem ser definidos. As operações lógico/matemáticas a serem executadas sobre tais tipos de dados definidos não são diretamente suportadas pelas linguagens de programação, desta forma, funções devem ser definidas para a correta e efetiva manipulação de tais tipos de dados. A agregação dos tipos de dados definidos acrescida de suas funções relacionadas especificamente desenvolvidas para manipular tais estruturas é chamada de tipo abstrato de dados.

TADs permitem o desenvolvimento modular de aplicações, fato que propicia uma maneira organizada e eficiente para a criação de programas.

```
1 // Estrutura geral de uma TAD
2
3 //TAD: <Nome_da_TAD>
4
5 //Tipo ou estrutura de dados
6
7 //funções que manipulam a estrutura de dados acima definida
```

O tipo ou estrutura de dados deve ser definido no início do arquivo (biblioteca) que define a TAD de modo que todo o restante do módulo conheça a estrutura a ser manipulada.

Podemos utilizar a palavra reservada **typedef** para definir um tipo de dados customizado para a TAD sendo desenvolvida.

```
1 // Forma geral da definição typedef
2
3 typedef <tipo> nome;
```

As funções que compõem a TAD e manipulam a estrutura de dados especifica devem possuir a seguinte forma:

```
1 // Forma geral da definição typedef
2
3 <tipo_de_retorno> <nome_da_funcao>(<estrutura_de_dados_especifica>, <outros argumentos>)
```

Organizando TADs em Módulos

TADs normalmente são organizados em arquivos separados do programa principal, também conhecidos como módulos. Pode-se criar um arquivo “.h” contendo todas as funções e tipos de dados necessários para a implementação da TAD ou um arquivo “.h” contendo apenas os protótipos das funções e os tipos de dados a serem manipulados.

Exemplo de TAD

```
 1 /*
 2  * fracao.h
 3  *
 4  * Desc: TAD implementando o tipo de
 5  *       dados fracao
 6  *
 7  */
 8
 9 // Definicao da informacao a ser manipulada
10
11 typedef int fracao[2];
12
13 /*
14  * Definicao das funcoes que manipulam essa
15  * estrutura de dados
16  */
17
18 double retornaValorReal(fracao f)
19 {
20     return f[0] / f[1];
21 }
22
23 fracao somaFracao(fracao f1, f2)
24 {
25     fracao f;
26     f[1] = mmc(f1[1], f2[1]);
27     f[0] = f1[0] + f2[0];
28     return f;
29 }
30
31 fracao multiplicaFracao(fracao f1, f2)
32 {
33     fracao f;
34     f[1] = f1[1] * f2[1];
35     f[0] = f1[0] * f2[0];
36     return f;
37 }
38
```

```c
int mmc(int n1, int n2)
{
    // codigo para o calculo do mmc
}
```

TAD Ponto "Cabeçalho" - ponto.h

```c
#ifndef PONTO_H
#define PONTO_H

/*
 * TAD: Ponto (x,y)
 * Tipo exportado
 */

typedef struct ponto Ponto;

/*
 * Funções exportadas
 * Função cria
 * Aloca e retorna um ponto com coordenadas (x,y)
 */

Ponto* cria (float x, float y);

/*
 * Função libera
 * Libera a memória de um ponto previamente criado.
 */

void libera (Ponto* p);

/*
 * Função acessa
 * Devolve os valores das coordenadas de um ponto
 */

void acessa (Ponto* p, float* x, float* y);

/*
 * Função atribui
 * Atribui novos valores às coordenadas de um ponto
 */

void atribui (Ponto* p, float x, float y);

/*
 * Soma dois pontos
 *
 */

Ponto* soma (Ponto* p1, Ponto* p2);

/*
 * Função distancia
 * Retorna a distância entre dois pontos
 */

float distancia (Ponto* p1, Ponto* p2);

#endif // PONTO_H
```

TAD Ponto "Cabeçalho" - ponto.c

```c
#include <stdlib.h> /* malloc, free, exit */
#include <stdio.h> /* printf */
#include <math.h> /* sqrt */
#include "ponto.h"

struct ponto
{
```

```
 8     float x;
 9     float y;
10 };
11
12 Ponto* cria (float x, float y)
13 {
14     Ponto* p = (Ponto*) malloc(sizeof(Ponto));
15     if (p == NULL)
16     {
17         printf("Memória insuficiente!\n");
18         exit(1);
19     }
20     p->x = x;
21     p->y = y;
22     return p;
23 }
24
25 void libera (Ponto* p)
26 {
27     free(p);
28 }
29
30 void acessa (Ponto* p, float* x, float* y)
31 {
32     *x = p->x;
33     *y = p->y;
34 }
35
36 void atribui (Ponto* p, float x, float y)
37 {
38     p->x = x;
39     p->y = y;
40 }
41
42 Ponto* soma (Ponto* p1, Ponto* p2)
43 {
44     if (!p1 || !p2) return NULL;
45     return cria(p1->x + p2->x, p1->y + p2->y);
46 }
47
48 float distancia (Ponto* p1, Ponto* p2)
49 {
50     float dx = p2->x - p1->x;
51     float dy = p2->y - p1->y;
52     return sqrt (dx * dx + dy * dy);
53 }
```

Main

```
 1 #include <stdio.h>
 2 #include "ponto.h"
 3
 4 int main()
 5 {
 6     float x, y;
 7
 8     Ponto *p1 = cria(3.0, 5.0);
 9     acessa(p1, &x, &y);
10     printf("x = %.2f e y = %.2f\n", x, y);
11
12     Ponto *p2 = cria(4.0, 1.0);
13     Ponto *ps = soma(p1, p2);
14     acessa(ps, &x, &y);
15     printf("x = %.2f e y = %.2f\n", x, y);
16
17     libera(p1);
18     libera(p2);
19     libera(ps);
20     return 0;
21 }
```

Compilar

```
gcc -o pontos ponto.c main.c -lm
```

Exercícios

1. Quais são as duas partes constituintes necessárias para a definição de uma TAD;
2. Toda função que compõe uma TAD deve receber necessariamente pelo menos um atributo. Qual é este atributo, justifique sua resposta;
3. Escreva um programa que faça uso do TAD Ponto definida anteriormente;
4. crescente novas operações ao TAD Ponto, tais como soma e subtração de pontos;
5. Crie uma TAD para a manipulação de vetores;
6. Crie uma TAD para manipulação de strings;
7. Crie uma TAD para manipulação de números complexos;

Extra: Como deve ser feita a alocação de memória de uma TAD? Variáveis devem ser declaradas em tempo de compilação ou em tempo de execução via alocação dinâmica de memória? Justifique sua resposta e dê exemplos.

Roteiro 1

- Baixar os códigos de um exemplo de utilização de TAD (https://saulo.arisa.com.br/aulas/das5102/20122_aulaTAD_e01.zip);
- Gerar uma aplicação diferente, utilizando a biblioteca *list.h*;
- É proibido alterar os arquivos *list.c* e *list.h*;
- Comando para compilação:

```
user@host:~$ gcc -o films3 films3.c lista.c </syntaxhighlight>
```

Roteiro 2

Material de exercícios mais antigos. Pode ser utilizado como exercícios de reforço.

Objetivos

- Revisar os conceitos vistos na aula teórica relativos à tipos abstratos de dados;
- Revisar a definição de módulos usando arquivos ".h" e ".c"

Exercício 1

Crie uma TAD para manipulação de vetores de doubles

- Como estrutura de dados para esta TAD utilize:

```
1 typedef struct vetor_
2 {
3    int tamanho;
4    double *elementos;
5 } Vetor;
```

- A TAD deve permitir a manipulação de vetores de tamanho variável, ou seja, o tamanho do vetor a ser manipulado deve ser informado pelo usuário e alocado dinamicamente. A função *Vetor* criaVetor(int N)* deve se responsabilizar por tal criação.
- A função *liberaVetor(Vetor* v)* deve liberar a memória utilizada por v;
- Crie uma função que some todos os elementos do vetor;
- Crie a função *multiplicaVetor(Vetor* v, int N)* que multiplica todos os elementos do vetor pelo elemento N;
- Crie a função *quadradosVetor(Vetor* v)* que eleva todos os elementos do vetor ao quadrado;
- Crie a função vetor *somaVetores(Vetor* v1, Vetor* v2)* que deve testar se os dois vetores possuem o mesmo tamanho e em caso afirmativo somar os elementos do vetor 1 aos do vetor 2;

Esta página foi modificada pela última vez em 26 de novembro de 2016, às 15h26min.