9. Tipos Abstratos de Dados

R. Cerqueira, W. Celes e J.L. Rangel

Neste capítulo, discutiremos uma importante técnica de programação baseada na definição de *Tipos Abstratos de Dados* (TAD). Veremos também como a linguagem C pode nos ajudar na implementação de um TAD, através de alguns de seus mecanismos básicos de *modularização* (divisão de um programa em vários arquivos fontes).

9.1. Módulos e Compilação em Separado

Como foi visto no capítulo 1, um programa em C pode ser dividido em vários arquivos fontes (arquivos com extensão `.c`). Quando temos um arquivo com funções que representam apenas parte da implementação de um programa completo, denominamos esse arquivo de *módulo*. Assim, a implementação de um programa pode ser composta por um ou mais módulos.

No caso de um programa composto por vários módulos, cada um desses módulos deve ser compilado separadamente, gerando um arquivo objeto (geralmente um arquivo com extensão `.o` ou `.obj`) para cada módulo. Após a compilação de todos os módulos, uma outra ferramenta, denominada *ligador*, é usada para juntar todos os arquivos objeto em um único arquivo executável.

Para programas pequenos, o uso de vários módulos pode não se justificar. Mas para programas de médio e grande porte, a sua divisão em vários módulos é uma técnica fundamental, pois facilita a divisão de uma tarefa maior e mais complexa em tarefas menores e, provavelmente, mais fáceis de implementar e de testar. Além disso, um módulo com funções C pode ser utilizado para compor vários programas, e assim poupar muito tempo de programação.

Para ilustrar o uso de módulos em C, considere que temos um arquivo `str.c` que contém apenas a implementação das funções de manipulação de strings `comprimento`, `copia` e `concatena` vistas no capítulo 6. Considere também que temos um arquivo `prog1.c` com o seguinte código:

```
#include <stdio.h>

int comprimento (char* str);
void copia (char* dest, char* orig);
void concatena (char* dest, char* orig);

int main (void) {
  char str[101], str1[51], str2[51];
  printf("Entre com uma seqüência de caracteres: ");
  scanf(" %50[^\n]", str1);
  printf("Entre com outra seqüência de caracteres: ");
  scanf(" %50[^\n]", str2);
  copia(str, str1);
  concatena(str, str2);
  printf("Comprimento da concatenação: %d\n",comprimento(str));
  return 0;
}
```

A partir desses dois arquivos fontes, podemos gerar um programa executável compilando cada um dos arquivos separadamente e depois ligando-os em um único

arquivo executável. Por exemplo, com o compilador Gnu C (gcc) utilizaríamos a seguinte seqüência de comandos para gerar o arquivo executável `prog1.exe`:

```
> gcc –c str.c
> gcc –c prog1.c
> gcc –o prog1.exe str.o prog1.o
```

O mesmo arquivo `str.c` pode ser usado para compor outros programas que queiram utilizar suas funções. Para que as funções implementadas em `str.c` possam ser usadas por um outro módulo C, este precisa conhecer os cabeçalhos das funções oferecidas por `str.c`. No exemplo anterior, isso foi resolvido pela repetição dos cabeçalhos das funções no início do arquivo `prog1.c`. Entretanto, para módulos que ofereçam várias funções ou que queiram usar funções de muitos outros módulos, essa repetição manual pode ficar muito trabalhosa e sensível a erros. Para contornar esse problema, todo módulo de funções C costuma ter associado a ele um arquivo que contém apenas os cabeçalhos das funções oferecidas pelo módulo e, eventualmente, os tipos de dados que ele exporte (`typedef`'s, `struct`'s, etc). Esse arquivo de cabeçalhos segue o mesmo nome do módulo ao qual está associado, só que com a extensão `.h`. Assim, poderíamos definir um arquivo `str.h` para o módulo do exemplo anterior, com o seguinte conteúdo:

```
/* Funções oferecidas pelo modulo str.c */

/* Função comprimento
** Retorna o número de caracteres da string passada como parâmetro
*/
int comprimento (char* str);

/* Função copia
** Copia os caracteres da string orig (origem) para dest (destino)
*/
void copia (char* dest, char* orig);

/* Função concatena
** Concatena a string orig (origem) na string dest (destino)
*/
void concatena (char* dest, char* orig);
```

Observe que colocamos vários comentários no arquivo `str.h`. Isso é uma prática muito comum, e tem como finalidade documentar as funções oferecidas por um módulo. Esses comentários devem esclarecer qual é o comportamento esperado das funções exportadas por um módulo, facilitando o seu uso por outros programadores (ou pelo mesmo programador algum tempo depois da criação do módulo).

Agora, ao invés de repetir manualmente os cabeçalhos dessas funções, todo módulo que quiser usar as funções de `str.c` precisa apenas incluir o arquivo `str.h`. No exemplo anterior, o módulo `prog1.c` poderia ser simplificado da seguinte forma:

```
#include <stdio.h>
#include "str.h"

int main (void) {
  char str[101], str1[51], str2[51];
  printf("Entre com uma seqüência de caracteres: ");
  scanf(" %50[^\n]", str1);
  printf("Entre com outra seqüência de caracteres: ");
  scanf(" %50[^\n]", str2);
```

```
    copia(str, str1);
    concatena(str, str2);
    printf("Comprimento da concatenação: %d\n",comprimento(str));
    return 0;
}
```

Note que os arquivos de cabeçalhos das funções da biblioteca padrão do C (que acompanham seu compilador) são incluídos da forma `#include <arquivo.h>`, enquanto que os arquivos de cabeçalhos dos seus módulos são geralmente incluídos da forma `#include "arquivo.h"`. O uso dos delimitadores `< >` e `" "` indica para o compilador onde ele deve procurar esses arquivos de cabeçalhos durante a compilação.

9.2. Tipo Abstrato de Dados

Geralmente, um módulo agrupa vários tipos e funções com funcionalidades relacionadas, caracterizando assim uma finalidade bem definida. Por exemplo, na seção anterior vimos um módulo com funções para manipulação de cadeias de caracteres. Nos casos em que um módulo define um novo tipo de dado e o conjunto de operações para manipular dados desse tipo, falamos que o módulo representa um *tipo abstrato de dados* (TAD). Nesse contexto, *abstrato* significa "esquecida a forma de implementação", ou seja, um TAD é descrito pela finalidade do tipo e de suas operações, e não pela forma como está implementado.

Podemos, por exemplo, criar um TAD para representar matrizes alocadas dinamicamente. Para isso, criamos um tipo "matriz" e uma série de funções que o manipulam. Podemos pensar, por exemplo, em funções que acessem e manipulem os valores dos elementos da matriz. Criando um *tipo abstrato*, podemos "esconder" a estratégia de implementação. Quem usa o tipo abstrato precisa apenas conhecer a funcionalidade que ele implementa, não a forma como ele é implementado. Isto facilita a manutenção e o re-uso de códigos.

O uso de módulos e TADs são técnicas de programação muito importantes. Nos próximos capítulos, vamos procurar dividir nossos exemplos e programas em módulos e usar tipos abstratos de dados sempre que isso for possível. Antes disso, vamos ver alguns exemplos completos de TADs.

Exemplo 1: TAD Ponto

Como nosso primeiro exemplo de TAD, vamos considerar a criação de um tipo de dado para representar um ponto no R^2. Para isso, devemos definir um tipo abstrato, que denominaremos de `Ponto`, e o conjunto de funções que operam sobre esse tipo. Neste exemplo, vamos considerar as seguintes operações:

- `cria`: operação que cria um ponto com coordenadas `x` e `y`;
- `libera`: operação que libera a memória alocada por um ponto;
- `acessa`: operação que devolve as coordenadas de um ponto;
- `atribui`: operação que atribui novos valores às coordenadas de um ponto;
- `distancia`: operação que calcula a distância entre dois pontos.

A interface desse módulo pode ser dada pelo código a seguir:

Arquivo 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);

/* Função distancia
** Retorna a distância entre dois pontos
*/
float distancia (Ponto* p1, Ponto* p2);
```

Note que a composição da estrutura `Ponto` (`struct ponto`) não é exportada pelo módulo. Dessa forma, os demais módulos que usarem esse TAD não poderão acessar diretamente os campos dessa estrutura. Os *clientes* desse TAD só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo `ponto.h`.

Agora, mostraremos uma implementação para esse tipo abstrato de dados. O arquivo de implementação do módulo (arquivo `ponto.c`) deve sempre incluir o arquivo de interface do módulo. Isto é necessário por duas razões. Primeiro, podem existir definições na interface que são necessárias na implementação. No nosso caso, por exemplo, precisamos da definição do tipo `Ponto`. A segunda razão é garantirmos que as funções implementadas correspondem às funções da interface. Como o protótipo das funções exportadas é incluído, o compilador verifica, por exemplo, se os parâmetros das funções implementadas equivalem aos parâmetros dos protótipos. Além da própria interface, precisamos naturalmente incluir as interfaces das funções que usamos da biblioteca padrão.

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

Como só precisamos guardar as coordenadas de um ponto, podemos definir a estrutura `ponto` da seguinte forma:

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

A função que cria um ponto dinamicamente deve alocar a estrutura que representa o ponto e inicializar os seus campos:

```
Ponto* cria (float x, float y) {
   Ponto* p = (Ponto*) malloc(sizeof(Ponto));
   if (p == NULL) {
      printf("Memória insuficiente!\n");
      exit(1);
   }
   p->x = x;
   p->y = y;
   return p;
}
```

Para esse TAD, a função que libera um ponto deve apenas liberar a estrutura que foi criada dinamicamente através da função `cria`:

```
void libera (Ponto* p) {
   free(p);
}
```

As funções para acessar e atribuir valores às coordenadas de um ponto são de fácil implementação, como pode ser visto a seguir.

```
void acessa (Ponto* p, float* x, float* y) {
   *x = p->x;
   *y = p->y;
}

void atribui (Ponto* p, float x, float y) {
   p->x = x;
   p->y = y;
}
```

Já a operação para calcular a distância entre dois pontos pode ser implementada da seguinte forma:

```
float distancia (Ponto* p1, Ponto* p2) {
   float dx = p2->x - p1->x;
   float dy = p2->y - p1->y;
   return sqrt(dx*dx + dy*dy);
}
```

Exercício: Escreva um programa que faça uso do TAD ponto definido acima.

Exercício: Acrescente novas operações ao TAD ponto, tais como soma e subtração de pontos.

Exercício: Acrescente novas operações ao TAD ponto, de tal forma que seja possível obter uma representação do ponto em coordenadas polares.

Exercício: Implemente um novo TAD para representar pontos no R^3.

Exemplo 2: TAD Matriz

Como foi discutido anteriormente, a implementação de um TAD fica "escondida" dentro de seu módulo. Assim, podemos experimentar diferentes maneiras de implementar um mesmo TAD, sem que isso afete os seus clientes. Para ilustrar essa independência de implementação, vamos considerar a criação de um tipo abstrato de dados para representar matrizes de valores reais alocadas dinamicamente, com dimensões `m` por `n` fornecidas em tempo de execução. Para tanto, devemos definir um tipo abstrato, que denominaremos de `Matriz`, e o conjunto de funções que operam sobre esse tipo. Neste exemplo, vamos considerar as seguintes operações:

- `cria`: operação que cria uma matriz de dimensão `m` por `n`;
- `libera`: operação que libera a memória alocada para a matriz;
- `acessa`: operação que acessa o elemento da linha `i` e da coluna `j` da matriz;
- `atribui`: operação que atribui o elemento da linha `i` e da coluna `j` da matriz;
- `linhas`: operação que devolve o número de linhas da matriz;
- `colunas`: operação que devolve o número de colunas da matriz.

A interface do módulo pode ser dada pelo código abaixo:

Arquivo matriz.h:

```
/* TAD: matriz m por n */

/* Tipo exportado */
typedef struct matriz Matriz;


/* Funções exportadas */

/* Função cria
** Aloca e retorna uma matriz de dimensão m por n
*/
Matriz* cria (int m, int n);

/* Função libera
** Libera a memória de uma matriz previamente criada.
*/
void libera (Matriz* mat);

/* Função acessa
** Retorna o valor do elemento da linha i e coluna j da matriz
*/
float acessa (Matriz* mat, int i, int j);

/* Função atribui
** Atribui o valor dado ao elemento da linha i e coluna j da matriz
*/
void atribui (Matriz* mat, int i, int j, float v);

/* Função linhas
** Retorna o número de linhas da matriz
*/
int linhas (Matriz* mat);
```

```
/* Função colunas
** Retorna o número de colunas da matriz
*/
int colunas (Matriz* mat);
```

A seguir, mostraremos a implementação deste tipo abstrato usando as duas estratégias apresentadas no capítulo 8: matrizes dinâmicas representadas por vetores simples e matrizes dinâmicas representadas por vetores de ponteiros. A interface do módulo independe da estratégia de implementação adotada, o que é altamente desejável, pois podemos mudar a implementação sem afetar as aplicações que fazem uso do tipo abstrato. O arquivo `matriz1.c` apresenta a implementação através de vetor simples e o arquivo `matriz2.c` apresenta a implementação através de vetor de ponteiros.

Arquivo matriz1.c:

```
#include <stdlib.h>      /* malloc, free, exit */
#include <stdio.h>       /* printf */
#include "matriz.h"

struct matriz {
   int lin;
   int col;
   float* v;
};


Matriz* cria (int m, int n) {
   Matriz* mat = (Matriz*) malloc(sizeof(Matriz));
   if (mat == NULL) {
      printf("Memória insuficiente!\n");
      exit(1);
   }
   mat->lin = m;
   mat->col = n;
   mat->v = (float*) malloc(m*n*sizeof(float));
   return mat;
}

void libera (Matriz* mat){
   free(mat->v);
   free(mat);
}

float acessa (Matriz* mat, int i, int j) {
   int k;    /* índice do elemento no vetor */

   if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
      printf("Acesso inválido!\n");
      exit(1);
   }
   k = i*mat->col + j;
   return mat->v[k];
}

void atribui (Matriz* mat, int i, int j, float v) {
   int k;    /* índice do elemento no vetor */

   if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
      printf("Atribuição inválida!\n");
      exit(1);
   }
```

```
   k = i*mat->col + j;
   mat->v[k] = v;
}

int linhas (Matriz* mat) {
   return mat->lin;
}

int colunas (Matriz* mat) {
   return mat->col;
}
```

Arquivo matriz2.c:

```
#include <stdlib.h>      /* malloc, free, exit */
#include <stdio.h>       /* printf */
#include "matriz.h"

struct matriz {
   int lin;
   int col;
   float** v;
};

Matriz* cria (int m, int n) {
   int i;
   Matriz* mat = (Matriz*) malloc(sizeof(Matriz));
   if (mat == NULL) {
      printf("Memória insuficiente!\n");
      exit(1);
   }
   mat->lin = m;
   mat->col = n;
   mat->v = (float**) malloc(m*sizeof(float*));
   for (i=0; i<m; i++)
      mat->v[i] = (float*) malloc(n*sizeof(float));
   return mat;
}

void libera (Matriz* mat) {
   int i;
   for (i=0; i<mat->lin; i++)
      free(mat->v[i]);
   free(mat->v);
   free(mat);
}

float acessa (Matriz* mat, int i, int j) {
   if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
      printf("Acesso inválido!\n");
      exit(1);
   }
   return mat->v[i][j];
}

void atribui (Matriz* mat, int i, int j, float v) {
   if (i<0 || i>=mat->lin || j<0 || j>=mat->col) {
      printf("Atribuição inválida!\n");
      exit(1);
   }
   mat->v[i][j] = v;
}

int linhas (Matriz* mat) {
   return mat->lin;
}
```

```
int colunas (Matriz* mat) {
   return mat->col;
}
```

Exercício: Escreva um programa que faça uso do TAD matriz definido acima. Teste o seu programa com as duas implementações vistas.

Exercício: Usando apenas as operações definidas pelo TAD matriz, implemente uma função que determine se uma matriz é ou não quadrada simétrica.

Exercício: Usando apenas as operações definidas pelo TAD matriz, implemente uma função que, dada uma matriz, crie dinamicamente a matriz transposta correspondente.

Exercício: Defina um TAD para implementar matrizes quadradas simétricas, de acordo com a representação sugerida no capítulo 8.