Tipos Abstratos de Dados

Atílio G. Luiz

19 de setembro de 2023

Tipo abstrato de dado (TAD) é uma especificação de um conjunto de dados e operações que podem ser executadas sobre esses dados.

A ideia que fundamenta os tipos abstratos de dados é a de que os tipos nativos (int, float, etc.) 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 esses novos tipos de dados definidos não são diretamente suportadas pelas linguagens de programação. Deste modo, 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 (TAD)**.

Encapsulamento e abstração de dados

Uma das ideias centrais no desenvolvimento de um TAD é conseguir esconder a forma concreta com que ele foi implementado (encapsulamento) e fornecer apenas o essencial (interface) e ocultar os detalhes (abstração) do usuário do seu TAD.

Uma analogia que podemos fazer aqui é com um controle remoto (Figura 1). Ao usar um controle remoto de televisão, não precisamos saber como os circuitos internos do controle se interrelacionam para poder operar a TV. Precisamos apenas entender o que cada botão no controle faz ao ser apertado, ou seja, precisamos conhecer apenas a interface do controle remoto. Os demais dados técnicos são ocultados do usuário e o que fica visível é apenas um conjunto de funcionalidades apropriadas para o manuseio do dispositivo.

Figura 1: Não precisamos entender o funcionamento interno do controle remoto para podermos usá-lo.

Exemplo: TAD Ponto

Como exemplo de TAD, vamos considerar a criação de um tipo de dado para representar um ponto no $\mathbb{R}^2$. Para isso, devemos definir um tipo abstrato, denominado **Ponto**, e o conjunto de operações que operam sobre esse tipo.

Neste exemplo, vamos considerar as seguintes operações:

- `cria`: operação que cria um ponto com coordendadas x e y;
- `libera`: operação que libera a memória alocada por um ponto;
- `acessa x`: operação que retorna a coordenada x do ponto;
- `acessa y`: operação que retorna a coordenada y do ponto;
- `atribui x`: operação que atribui um novo valor à coordenada x do ponto;
- `atribui y`: operação que atribui um novo valor à coordenada y do ponto;
- `distancia`: operação que calcula a distância entre dois pontos;

Em C++, um TAD pode ser naturalmente implementado utilizando o conceito de **classe** da programação orientada a objetos. Assim, criaremos uma classe chamada **Ponto**. Além disso, podemos utilizar também a técnica de modularização a fim de separar a interface do TAD da sua implementação. Desta forma, o TAD Ponto pode ser implementado em dois arquivos separados:

- `Ponto.h`: arquivo de cabeçalho que contém apenas a declaração da classe Ponto;
- `Ponto.cpp`: arquivo-fonte que contém a implementação das funções-membro da classe Ponto.

A interface desse módulo pode ser dada pelo arquivo `Ponto.h` ilustrado a seguir:

```cpp
// Arquivo Ponto.h
#ifndef PONTO_H
#define PONTO_H

/**
 * TAD Ponto definido como uma classe que contem dados e operacoes
 * encapsulados. Todos os dados sao privados e as operacoes
 * sao publicas e sao acessiveis de fora da classe.
 */
class Ponto {
private:
    double x; // coordenada x
    double y; // coordenada y

public:
    // função construtora: é chamada quando o Ponto2 é criado
    Ponto(double valueX, double valueY);

    // retorna o valor da coordenada x
    double getX();

    // retorna o valor da coordenada y
    double getY();

    // atribui novo valor a coordenada x
    void setX(double newX);

    // atribui novo valor a coordenada y
    void setY(double newY);

    // calcula e retorna a distancia entre este ponto e o ponto q
    double distancia(Ponto *q);

    // retorna uma string contendo as coordenadas do ponto
    std::string toString();
};
```

```
37
38  #endif
```

Se conhecermos apenas a interface do TAD, podemos criar programas que usem as funcionalidades da classe. O arquivo que usa o TAD deve, obrigatoriamente, incluir o arquivo de cabeçalho responsável por definir sua interface. Por exemplo, logo a seguir mostramos um exemplo de programa-cliente que usa o TAD Ponto:

```
1   /**
2    * Arquivo Main.cpp
3    */
4   #include <iostream>
5   #include "Ponto.h"
6
7   using namespace std;
8
9   int main()
10  {
11          // cria dois pontos alocando memória dinamicamente
12          Ponto *p = new Ponto(3,4);
13          Ponto *q = new Ponto(5,6);
14
15          // Imprime os pontos na tela
16          cout << p->toString() << endl;
17          cout << q->toString() << endl;
18
19          // Calcula a distancia entre os dois pontos e imprime
20          cout << "distancia = " << p->distancia(q) << endl;
21
22          // libera a memória que foi alocada para os dois pontos
23          delete p;
24          delete q;
25
26          return 0;
27  }
```

A seguir, mostramos uma implementação para o TAD Ponto. O arquivo de implementação do módulo (arquivo `Ponto.cpp`) deve sempre incluir o arquivo de interface do módulo. Uma razão para isso é garantir que as funções-membro implementadas correspondam às funções-membro da classe da interface. A seguir, o arquivo-fonte `Ponto.cpp` é exibido:

```
1   /**
2    * Arquivo Ponto.cpp
3    */
4   #include <cmath>
5   #include <sstream>
6   #include "Ponto.h"
7
8   Ponto::Ponto(double valueX, double valueY) {
9       x = valueX;
10      y = valueY;
11  }
12
13  double Ponto::getX() {
14      return x;
15  }
16
```

```cpp
double Ponto::getY() {
    return y;
}

void Ponto::setX(double newX) {
    x = newX;
}

void Ponto::setY(double newY) {
    y = newY;
}

double Ponto::distancia(Ponto *q) {
    double dx = pow(q->x - x, 2);
    double dy = pow(q->y - y, 2);
    return sqrt(dx + dy);
}

std::string Ponto::toString() {
    std::stringstream sx, sy;
    sx << x;
    sy << y;
    return "(" + sx.str() + "," + sy.str() + ")";
}
```

Compilação em separado

As classes C++ são normalmente divididas em dois arquivos. O primeiro é o **arquivos de cabeçalho**, que possui a extensão .h e contém a declaração da classe e das funções-membro da classe. A implementação das funções-membro vão para o arquivo com a extensão .cpp, que é o **arquivo-fonte**. Fazendo isso, se a implementação da sua classe não for alterada então ela não precisa ser recompilada.

No exemplo acima, temos três arquivos:

- `Main.cpp`: arquivo que utiliza o TAD Ponto.
- `Ponto.h`: definição do TAD Ponto.
- `Ponto.cpp`: implementação do TAD Ponto.

A compilação do nosso projeto será executada nos seguintes passos:

1. Gerando o arquivo objeto Ponto.o

   ```
   g++ -c Ponto.cpp
   ```

2. Gerando o arquivo objeto Main.o

   ```
   g++ -c Main.cpp
   ```

3. Fazendo a linkagem e gerando o executável

   ```
   g++ Main.o Ponto.o -o main
   ```

Essas operações podem ser automatizadas usando um arquivo `Makefile`:

```
# Arquivo Makefile
main: Ponto.o Main.o
    g++ Main.o Ponto.o -o main
Ponto.o : Ponto.cpp
    g++ -c Ponto.cpp
```

```
6  Main.o : Main.cpp
7      g++ -c Main.cpp
```

Para usar o arquivo Makefile, basta acessar, via terminal, a pasta em que o arquivo está e digitar o comando `make`.