Tipos Abstratos de Dados

Rômulo César Silva

Unioeste

Abril de 2016

Sumário

Tipo de Dados

Um **tipo de dados** em uma linguagem de programação define o conjunto de valores (domínio) que uma variável, constante ou função pode assumir.

Em geral, as linguagens de programação têm tipos pré-definidos. Exemplos na linguagem C:

- `int`, `float`, `char`, ...

Exemplos em Pascal:

- `integer`, `real`, `boolean`, ...

Tipo de Dados

As linguagens de programação determinam quantos bits ou bytes devem ser ocupados na memória para cada tipo pré-definido.

Exemplo na linguagem C:

- `int`: 2 ou 4 bytes dependendo da arquitetura

Exemplo na linguagem Pascal:

- `boolean`: 1 byte

Tipo de Dados

Além disso, as linguagens de programação também definem o conjunto de operações aplicáveis a cada tipo de dados.

Exemplo na linguagem C:

- `int`: $\{+, -, *, \%\}$

Exemplo na linguagem Pascal:

- `boolean`: $\{\texttt{and}, \texttt{or}, \texttt{not}\}$

Tipo de Dados

Em geral o programador pode criar novos tipos de dados e também definir operações que podem ser aplicáveis aos tipos de dados criados.

Na linguagem C:

- uso de `struct`, `union`, `typedef`, vetores

```
typedef int inteiro;

struct aluno {
    int matricula;
    char nome[50];
    };
```

Tipo de Dados

Exemplo na linguagem Pascal:

- uso de `record`, `type`, `array`

```
type ponto = record
               x: real;
               y: real;
             end;

function calcula_distancia(p1,p2: ponto): real;
  begin
    calcula_distancia := sqrt(sqr(p2.x - p1.x) +
                              sqr(p2.y - p1.y));
  end;
```

Variações de Implementação

Pode-se implementar diferentemente o mesmo tipo de dados.
Ex.: implementação 1: usando estrutura

```
#include <math.h>

typedef struct {
      double x;
      double y;
      } ponto;

double distancia(ponto p1, ponto p2) {
   return sqrt(pow(p2.x - p1.x,2) +
                    pow(p2.y - p1.y,2));
}
```

Variações de Implementação

Pode-se implementar diferentemente o mesmo tipo de dados.
Ex.: implementação 2: usando vetor

```
#include <math.h>

typedef double ponto[2];

double distancia(ponto p1, ponto p2) {
   return sqrt(pow(p2[0] - p1[0],2) +
                          pow(p2[1] - p1[1],2));
}
```

Tipo Abstrato de Dados (TAD)

Um **Tipo Abstrato de Dados** (TAD) é um modelo matemático que encapsula a representação de dados e as operações que podem ser realizadas sobre eles.

- abstrai-se da representação interna usada em linguagens de programação.
- todas as diferentes implementações possíveis representam o mesmo domínio
- o significado das operações é o mesmo nas diferentes implementações

Tipo Abstrato de Dados (TAD)

No exemplo anterior do TAD `ponto`:

- o significado da operação `distancia` é o mesmo nas duas implementações
- além disso, o protótipo da operação `distancia` (tipos dos parâmetros e retorno) é o mesmo nas duas implementações, o que permite trocar uma representação pela outra sem necessidade de alteração em demais pontos do código.
- a rigor, o usuário do TAD `ponto` não precisa se preocupar sobre os detalhes internos da representação e implementação.

Tipo Abstrato de Dados (TAD)

Observações

- delimita intrinsicamente uma fronteira entre o programador do TAD e o usuário do TAD:
  - usuário não precisa conhecer detalhes (**o como**) da implementação, apenas **o que** faz cada operação
  - programador do TAD pode alterar a implementação interna sem afetar os usuários do TAD, desde que mantenha o significado de cada operação.
- melhora a qualidade do código: pode-se substituir uma parte do código sem afetar o restante.
- estimula a reutilização de código.

Tipo Abstrato de Dados (TAD)

- princípio básico usado no encapsulamento de linguagens orientadas a objetos
- a *abstração* consiste exatamente em não precisar conhecer os detalhes da implementação
- na linguagem C:
  - os arquivos **.h** contém a interface ou especificação do TAD
  - os arquivos **.c** contém a implementação do TAD
  - usuário do TAD precisa somente conhecer o **.h**

Tipo Abstrato de Dados (TAD)

Para cada operação deve-se especificar:

- Valores de **entrada** e **saída** esperados
- **Pré-condições**: propriedades dos valores de entrada assumidas pela operação. Ou seja, o resultado da operação só é garantido quando as pré-condições são satisfeitas.
- **Pós-condições**: efeitos causados pela execução da operação no caso das pré-condições terem sido atendidas.

Tipo Abstrato de Dados (TAD)

Geralmente:

- as entradas correspondem aos parâmetros de entrada
- as **saídas** correspondem aos parâmetros de saída (passados por referência) e/ou ao retorno da função

TAD - exemplo

arquivo .h:

```
typedef int TipoItem;
typedef struct no {
    TipoItem info;
    struct no * prox;
} Lista;

// Insere um elemento na cabeça da lista
// Pré-condição: nenhuma
// Pós-condição: retorna a cabeça da lista c/ o elemento inserido
Lista* insere(Lista* l, TipoItem info);

// Retira um elemento da lista
// Pré-condição: nenhuma
// Pós-condição: elemento é retirado da lista caso esteja presente
Lista * retira(Lista* l, TipoItem info);
```

Programação por Contrato

É uma metáfora em que os elementos de um software colaboram entre si, baseando-se em obrigações e benefícios mútuos, ao modo de um contrato, como no mundo dos negócios. Assim, o "Fornecedor" é o programador do TAD enquanto o "cliente" é o usuário do TAD. Assim, as pré-condições e pós-condições são as cláusulas contratuais [Meyer 1997].
De maneira resumida:

- **Pré-condições**: o que o cliente deve garantir antes de chamar a operação?
- **Pós-condições**: o que o fornecedor garante caso as pré-condições sejam atendidas?

Abstração

Em Algoritmos e Estrutura de Dados, o significado de abstração é:

> *a substituição de uma situação complexa e detalhada do mundo real por um modelo compreensível dentro do qual podemos solucionar um problema. Isto é, eliminam-se os detalhes cujo efeito sobre a solução do problema é mínimo ou não existente, portanto criando um modelo que nos permite lidar com a essência do problema*

[Aho and Ullman 1994].

Bibliografia I

[Aho and Ullman 1994] Alfred V. Aho; Jeffrey D. Ullman.
Foundations of Computer Science C Edition. 1994.

[Meyer 1997] Bertrand Meyer.
Object-Oriented Software Construction.
ISE Inc., 2nd Edition, 1997.