TADs
Tipos Abstratos de Dados

Wanderley de Souza Alencar
adaptado por Raphael Guedes

Universidade Federal de Goiás - UFG
Instituto de Informática - INF

15 de setembro de 2024

Introdução

Normalmente quando se inicia o estudo de *algoritmos* e/ou de *linguagens de programação*, vários conceitos são apresentados.

Um destes conceitos é o de dado.

O que é mesmo um *dado*?

Dado – Conceito

A palavra *dado* indica a representação de uma certa informação.

A informação é o resultado do tratamento dos dados que podem estar sob a forma de um texto, imagem, vídeo, áudio, etc.

No âmbito da Computação, dados são processados pela CPU do computador e armazenados em memória principal ou qualquer outro dispositivo de armazenamento secundário.

No nível mais elementar, os dados são simplesmente *cadeias binárias*.

Tipo de Dado

Outro conceito importante é o de tipo de dado.

O que é um *tipo de dado*?

Tipo de Dado – Conceito

Um *tipo de dado* define um conjunto de possíveis valores que uma certa variável pode assumir, bem como o conjunto de operações típicas, normalmente predefinidas, que podem ser realizadas com a variável (ou sobre a variável).

Tipo de Dado – Conceito

Por exemplo:

Se a variável X é do tipo `inteiro`, então espera-se que ela possa assumir valores no conjunto:

$$\mathbb{Z} = \{\ldots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \ldots\}$$

E que estejam disponíveis as operações de adição (+), subtração (-), multiplicação (x) e divisão inteira (/), pois elas são normalmente consideradas *operações típicas* sobre os números inteiros da Matemática.

Tipo de Dado – Conceito

Numa linguagem de programação específica, um mesmo tipo de dado pode ter o par (`valores`, `operações`) variando de acordo com o ambiente operacional em que ela (linguagem) está implementada.

Tipo de Dado – Conceito

Por exemplo:

No ambiente operacional A, a implementação dos inteiros para uma certa linguagem de programação $\mathbb{X}$ pode permitir números inteiros na faixa:

-2^{31} a $(2^{31} - 1)$

Já no ambiente operacional B, os inteiros de $\mathbb{X}$ estão na faixa:

-2^{63} a $(2^{63} - 1)$.

Tipo de Dado – Conceito

Enfatizando...

Um *tipo de dado* definirá um conjunto de

- valores (domínio) e de
- operações

associados a uma variável qualquer daquele tipo.

Tipo de Dado – Exemplo

Exemplo: **inteiro**

- domínio:
  - $< ? \ -2, -1, 0, +1, +2, \ ? >$
- operações:
  - adição;
  - subtração;
  - multiplicação;
  - divisão inteira;
  - ...

Tipo de Dado – Exemplo

Exemplo: **caractere**

- domínio
  - valores definidos por uma *tabela* de codificação (ASCII, Unicode, etc.).
- operações
  - comparação ($<, >, \leq, \geq, \neq, \ldots$);
  - conversão maiúscula/minúscula;
  - ...

Tipo de Dado – Exemplo

Tabela UNICODE

`https://pt.wikipedia.org/wiki/Unicode`

Tipo de Dado – Classificação

Nas linguagens de programação contemporâneas, normalmente um tipo de dado pode ser classificado como:

- Primitivo (básico, nativo, fundamental ou elementar);
- Estruturado (ou composto);
- Definido pelo usuário (ou construído).

Tipo de Dado – Primitivo

O tipo primitivo é aquele fornecido pela própria definição/implementação da linguagem de programação.

Portanto, os tipos primitivos formam um conjunto básico de tipos de dados disponíveis para o programador daquela linguagem.

Normalmente reflete elemento presentes no próprio *hardware* subjacente ao ambiente computacional no qual a linguagem está implementada. Algumas vezes incorporaram algum nível (pequeno) de abstração em relação a este *hardware*.

Tipo de Dado – Primitivo

São *normalmente* tipos primitivos:

- inteiro;
- real (número em *ponto flutuante*);
- complexo;
- caractere;
- lógico;
- ponteiro (ou apontador).

Tipo de Dado – Primitivo

Por exemplo, em $\mathbb{C}$:

- `short int` e `unsigned short int`;
- `int` e `unsigned int`;
- `long int` e `unsigned long int`;
- `long long int` e `unsigned long long int`;
- `float`;
- `double` e `long double`;
- `char` e `unsigned char`.

Tipo de Dado – Estruturado

O tipo estruturado é formado a partir da composição, ou agregação, de tipos primitivos ou de outros tipos estruturados previamente definidos.

Exige, por parte da linguagem que o possui, maior nível de abstração para sua implementação.

Ele permite, de maneira *elegante*, organizar os dados de modo que possam ser utilizados eficientemente, reunindo um conjunto de vários tipos de dados sob um único conceito.

Tipo de Dado – Estruturado

São *normalmente* tipos estruturados:

- cadeia de caracteres, ou *string*;
- registro;
- vetor;
- matriz.

Tipo de Dado – Estruturado

Por exemplo, em $\mathbb{C}$:

- `struct`;

Tipo de Dado – Estruturado

Exemplo em $\mathbb{C}$:

```
1  //
2  // Representacao de uma conta-corrente numa instituicao financeira
3  //
4  struct ContaCorrente {
5      unsigned int numero;
6      char * nomeTitular;
7      unsigned int telefoneTitular;
8      bool contaConjunta;
9      char * nomeDependente;
10     float saldoAtual;
11     bool estaAtiva;
12 }
```

Tipo de Dado – Definido pelo Usuário

Os tipos de dados presentes numa linguagem de programação podem não ser suficientes para o desenvolvimento de uma certa aplicação.

Assim convém termos a possibilidade de criar novos tipos de dados que, por consequência, são chamados de tipos definidos pelo usuário.

Tipo de Dado – Definido pelo Usuário

Por exemplo, em $\mathbb{C}$:

- `bool`;

Tipo de Dado – Definido pelo Usuário

Exemplo em $\mathbb{C}$:

```c
#include <stdio.h>
#include <stdbool.h>
//
// Declaracao de um tipo logico (boolean)
//
int main( ) {
   bool emCrash;

   emCrash = false;
   if (emCrash == true) {
      printf("Sistema em CRASH.\n");
   }
   else {
      printf("Sistema OK!\n");
   }
   return (0);
}
```

Tipo de Dado – Definido pelo Usuário

Há autores que consideram que, em $\mathbb{C}$, os tipos `struct` e `union` são também tipos definidos pelo usuário.

Em minha opinião eles são *tipos estruturados* e, por isso, assim os classifiquei.

Portanto ficará a seu critério adotar uma ou outra classificação.

Tipo de Dado – Definido pelo Usuário

Tipos de Dados		
Primitivo	Estruturado	Definido pelo Usuário
inteiro	string	enumerações
real	vetor	tuplas
complexo	matriz	listas
lógico	registro	
ponteiro		

Tabela: Tipos de dados em programação

TAD – Tipo Abstrato de Dado

TAD

Em muitas situações concebemos, mentalmente, novos *tipos de dados* e imaginamos *operações* sendo realizadas com eles, ou sobre eles.

Assim convém termos a possibilidade de criar novos tipos de dados, como também especificar quais serão as operações disponíveis para manipular variáveis destes *novos* tipos.

TAD – Tipo Abstrato de Dado

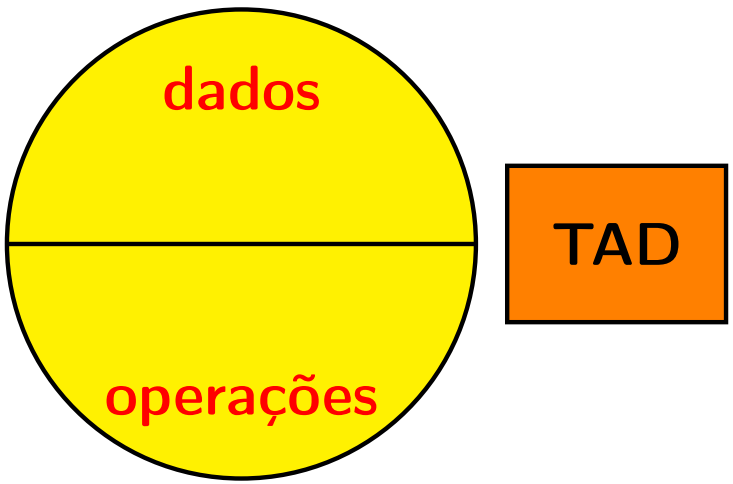
TAD

Estes *novos tipos*, com as suas *operações*, são chamados de **T**ipos **A**bstratos de **D**ados – TADs.

TAD – Tipo Abstrato de Dado

TAD – Conceito

Um TAD é uma maneira do programador definir um *novo tipo* de dados juntamente com as *operações* que manipularão este novo tipo.



TAD – Tipo Abstrato de Dado

O que é *abstração*?

"*Uma abstração é a visão, ou representação, de uma entidade que inclui apenas seus atribuitos mais significativos. No sentido geral, a abstração permite-nos colecionar instâncias de entidades em grupos nos quais seus atributos comuns não precisam ser considerados.*"

(SEBESTA, *Concepts of Programming Languages*, 2012, pp. 474).

TAD – Tipo Abstrato de Dado

O que é *abstração*?

A *abstração*, numa linguagem de programação, é uma arma contra a complexidade da própria programação, simplificando este processo.

Com o uso da *abstração* é possível concentrar-nos nos aspectos essenciais de um contexto qualquer, ignorando características que são somenos importantes, acidentais, acessórias.

O que é *abstração*?

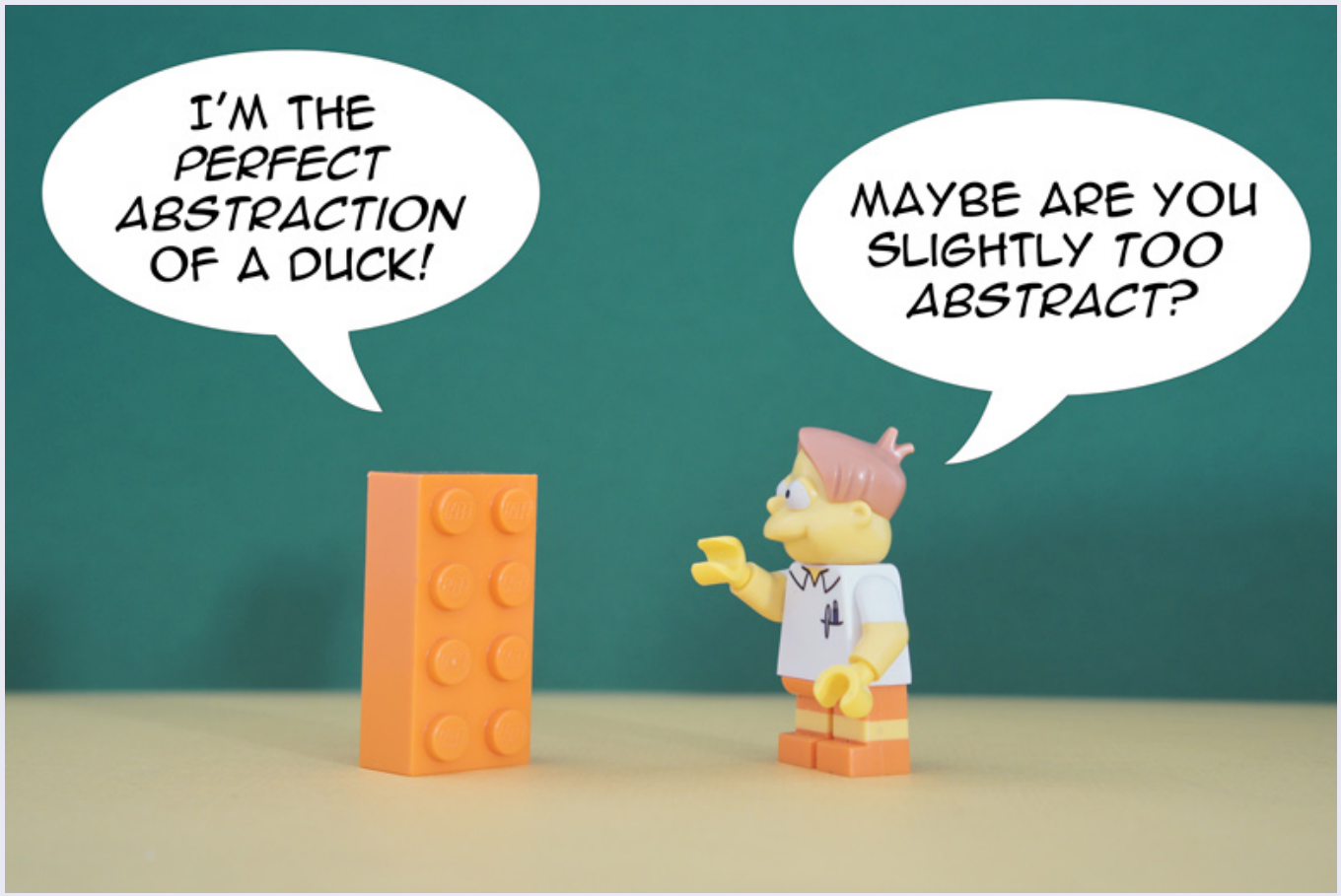


Figura: Um bloco de montar laranja dizendo que é um pato.

TAD – Tipo Abstrato de Dado

TAD – Conceito

Um Tipo Abstrato de Dado (TAD) é a especificação de um conjunto de dados e das operações que podem ser executadas sobre eles, independentemente da maneira como tudo isto será implementado.

O objetivo é proporcionar ao desenvolvedor uma *abstração* que facilite a concepção – e programação – de uma aplicação, pois as variáveis de um TAD podem ser tratadas como "*entidades fechadas*" (ou *caixas-pretas*).

TAD – Tipo Abstrato de Dado

Figura: Visão de *caixa preta*.

TAD – Tipo Abstrato de Dado

Figura: Visão INTERNA da *caixa preta*.

TAD – Tipo Abstrato de Dado

Conceito

Um TAD...

- estabelece o conceito de tipo de dado separado de sua representação.

- é definido como um modelo matemático por meio de um par (`valores`, `operações`) em que:
  - valores – conjunto de valores que podem ser assumidos por uma variável daquele tipo;
  - operações – conjunto de operações possíveis sobre uma variável daquele tipo.

TAD – Exemplo 01

O tipo *números reais* – $\Re$ – pode ser definido como um TAD por:

- `valores` – $\Re$
- `operações` – $\{+, -, *, /, =, \neq, <, \leq, >, \geq\}$

TAD – Tipo Abstrato de Dado

Vantagens

A definição de um TAD, permite:

- separação entre conceito (definição do tipo) e implementação das operações;
- limitar a visibilidade da estrutura interna do TAD;
- controlar a visibilidade das operações perante o usuário, que passa a ser cliente do TAD;
- limitar o acesso do cliente somente à forma abstrata do TAD.
- ...

TAD – Tipo Abstrato de Dado

Vantagens

A definição de um TAD, permite:

- que o código do cliente não dependa da implementação dele: o TAD é uma *caixa preta* sob a ótica do cliente;
- segurança: clientes não podem alterar a representação, pois não possuem acesso a ela;
- segurança: clientes não podem tornar os dados inconsistentes, pois não possuem acesso a eles.

TAD – Tipo Abstrato de Dado

Projeto

Algumas diretrizes para projetar um *bom* TAD são:

- escolher as operações adequadas, definindo claramente o comportamento de cada uma delas;
- projetar operações flexíveis e suficientemente abrangentes para os diversos contextos de uso do TAD;
- implementar eficientemente cada operação definida;
- reutilizar operações básicas para elaborar outras mais complexas.

Projeto

Escolher as operações adequadas significa:

- definir um *pequeno* número de operações;
- que o conjunto de operações deve ser suficiente para realizar as computações necessárias às aplicações que utilizarão o TAD;
- que cada operação deve ter um propósito bem definido, com comportamento constante e coerente.

TAD – Exemplo 02

Ponto 2D

Considere que numa certa aplicação da área de *Computação Gráfica* deseja-se ter um TAD que represente um *ponto* no espaço bidimensional.

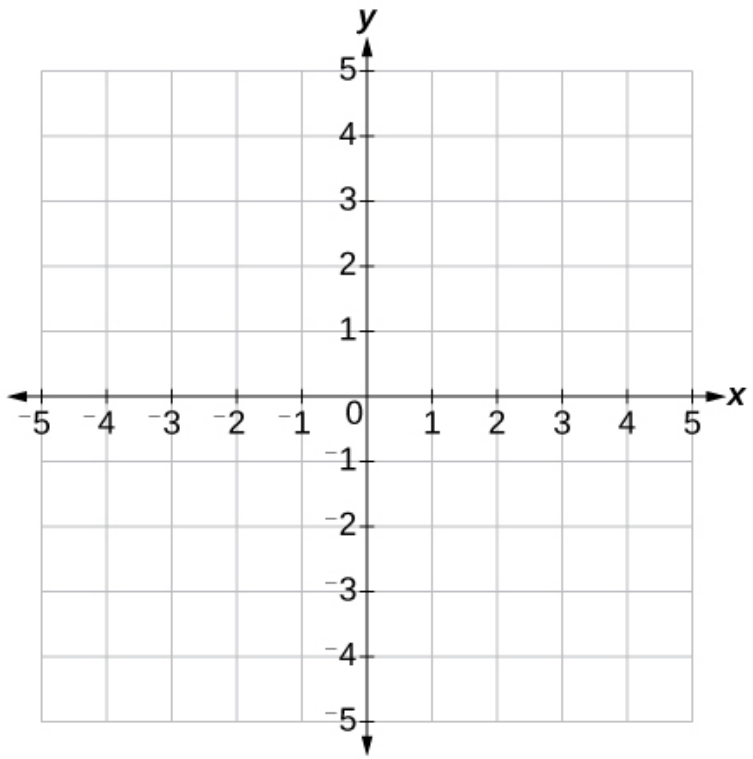


Figura: Plano cartesiano com 5 pontos em cada eixo.

TAD – Exemplo 02

Ponto 2D

Matematicamente um *ponto* no espaço bidimensional possui:

- Coordenada x – um número real ($x \in \Re$);
- Coordenada y – um número real ($y \in \Re$).

TAD – Exemplo 02

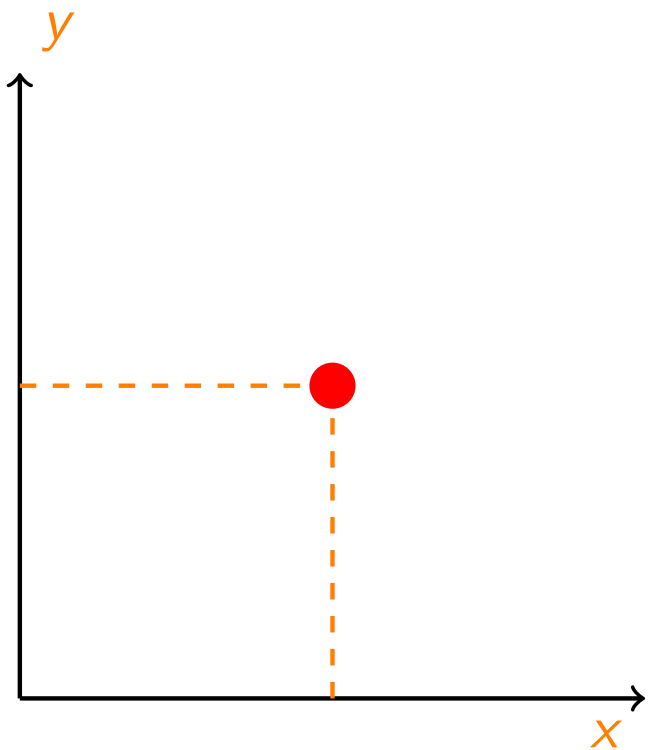


Figura: Um gráfico com os eixos X e Y e um ponto circular vermelho nas coordenadas X e Y

TAD – Exemplo 02

- Ponto (x,y)
  - Coordenada x – um número real ($x \in \Re$);
  - Coordenada y – um número real ($y \in \Re$).

- Par (v,o):
  - `valores` – dupla ordenada formada por dois reais: Ponto(x,y);
  - `operações` – operações aplicáveis sobre o tipo Ponto.

TAD – Exemplo 02

Operações

Quais operações são necessárias, neste contexto, em relação a um Ponto2D?

TAD – Exemplo 02

Operações

- *ponto_cria*: cria um ponto, alocando memória para as suas coordenadas;
- *ponto_libera* : libera a memória alocada por um ponto, destruindo-o;
- *ponto_acessa* : retorna as coordenadas de um ponto;
- *ponto_atribui* : atribui novos valores às coordenadas de um ponto;

TAD – Exemplo 02

Operações

- *ponto_distancia* : calcula a distância *euclidiana* entre dois pontos dados;
- *ponto_move*: move o ponto de uma posição para outra;
- *ponto_oculta* : torna o ponto *invisível*;
- *ponto_mostra* : torna o ponto *visível*;

TAD – Exemplo 02

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
//
// Definindo o tipo de dado
//
typedef struct ponto Ponto;

struct ponto {
    float x;
    float y;
    bool visibilidade; // true = visivel, false = invisivel
};
```

TAD – Exemplo 02

```
1  //
2  // Cria um ponto
3  //
4  Ponto* ponto_cria (float x, float y, bool visibilidade) {
5          Ponto* p = (Ponto*) malloc(sizeof(Ponto));
6          if (p != NULL) {
7                  p->x = x;
8                  p->y = y;
9                  p->visibilidade = visibilidade;
10         }
11         return p;
12 }
```

TAD – Exemplo 02

```
1 //
2 // Libera (desaloca) um ponto...
3 //
4 void ponto_libera (Ponto* p) {
5     if (p != NULL) {
6         free(p);
7     }
8 }
```

TAD – Exemplo 02

```
1 //
2 // Acessa um ponto, coletando suas coordenadas
3 //
4 void ponto_acessa (Ponto* p, float* x, float* y) {
5     if (p != NULL) {
6             *x=p->x;
7             *y=p->y;
8         }
9 }
```

TAD – Exemplo 02

```
1 //
2 // Atribui coordenadas a um ponto, modificando–o
3 //
4 void ponto_atribui (Ponto* p, float x, float y) {
5     if (p != NULL) {
6         p->x=x;
7         p->y=y;
8     }
9 }
```

TAD – Exemplo 02

```
1  //
2  // Retorna a distancia entre dois pontos
3  //
4  float ponto_distancia (Ponto* p1, Ponto* p2) {
5          float dx, dy;
6
7          dx = p1->x - p2->x;
8          dy = p1->y - p2->y;
9          return (sqrt(dx*dx+dy*dy));
10 }
```

TAD – Exemplo 02

```
1  //
2  // move o ponto para as coordenadas (x, y)
3  //
4  void ponto_move (Ponto* p, float x, float y, int movimento) {
5
6        //
7        // Move o ponto p, a partir de sua posicao atual, para
8        // a posicao (x, y) segundo um certo tipo de movimento.
9        //
10       // Por exemplo: 1 – linear
11       // 2 – zig–zag
12       // ...
13       //
14
15 }
```

TAD – Exemplo 02

```
1 //
2 // Oculta (torna invisivel) o ponto
3 //
4 void ponto_oculta (Ponto* p) {
5
6     p->visibilidade = false;
7 }
```

TAD – Exemplo 02

```
1 //
2 // Mostra (torna visivel) o ponto
3 //
4 void ponto_mostra (Ponto* p) {
5
6     p->visibilidade = true;
7 }
```

TAD – Exemplo 02

```
1  int main(){
2          float xp,yp,xq,yq,d;
3          Ponto *p,*q;
4
5          printf("digite as coordenadas x e y para o ponto 1: ");
6          scanf("%f %f",&xp,&yp);
7          printf("digite as coordenadas x e y para o ponto 2: ");
8          scanf("%f %f",&xq,&yq);
9          p = ponto_cria(xp,yp,true);
10         q = ponto_cria(xq,yq,true);
11         d = ponto_distancia(p,q);
12         ponto_acessa(p,&xp,&yp); ponto_acessa(q,&xq,&yq);
13         printf("Distancia entre os pontos (%.2f,%.2f) e (%.2f,%.2f) = %.5f\n",xp,
               yp,xq,yq,d);
14         ponto_libera(p); ponto_libera(q);
15         return (0);
16 }
```

TAD – Modularização e Implementação

Conceito

Sabemos que a definição de um TAD é conceitual: não há imposição quanto à implementação.

Cada linguagem de programação possui seus próprios padrões de como implementar um TAD.

TAD – Modularização e Implementação

Conceito

Sabemos, ainda, que num TAD:

- a definição do tipo de dado (ou sua representação) e de suas operações são contidas numa única unidade sintática;
- a representação deve ocultar detalhes da implementação, exibindo apenas as operações que estão disponíveis para manipular aquele tipo de dado.

TAD – Modularização e Implementação

Conceito

Vamos, agora, nos concentrar na modularização e implementação de um TAD em linguagem C.

Linguagem $\mathbb{C}$

Para criar um TAD na linguagem $\mathbb{C}$, convenciona-se preparar dois arquivos distintos:

- `tad.h`[a]
- `tad.c`

[a] O nome do arquivo pode ser livremente escolhido, desde que se utilize o mesmo nome para o arquivo **.h** e **.c**.

Linguagem $\mathbb{C}$

Na linguagem $\mathbb{C}$, convenciona-se preparar dois arquivos distintos:

- tad.h contém os *protótipos* das rotinas (procedimentos e funções), dos tipos ponteiro e de dados globalmente acessíveis à aplicação que utilizará o TAD.

Neste arquivo é definida a *interface* visível para o *cliente* do TAD.

Linguagem $\mathbb{C}$

Na linguagem $\mathbb{C}$, convenciona-se preparar dois arquivos:

- tad.c contém a declaração do tipo de dados e implementação das suas operações (procedimentos e funções).

O que for definido neste arquivo ficará invisível para o *cliente* do TAD.

Linguagem $\mathbb{C}$

- Dessa maneira separamos o "*conceito*" (definição do tipo) de sua "*implementação*".
- A esse processo de separação da definição do TAD em dois arquivos damos o nome de modularização.

TAD – Interface

Interface - TAD

É frequente, em Computação, que o termo `interface` se refira ao meio que o usuário utiliza para interagir com um sistema computacional:

- interface textual;
- interface gráfica;
- interface natural (por voz, por exemplo);
- etc.

Interface - TAD

No contexto dos TADs, o termo `interface` refere-se ao *protótipo* de uma função, ou seja, à declaração de uma função:

- `int factorial(int n);`
- `void swap(int * a; int * b);`
- `int * allocArray(int * v; int n);`

Interface - TAD

Por meio do uso de protótipos de função em arquivos de cabeçalho (.h) é possível especificar interfaces para bibliotecas de *software*.

Na interface pode-se especificar tipos que são globais e, portanto, acessíveis em todo o escopo da aplicação/módulo. Na interface podemos especificar ponteiros.

TAD – Exemplo 03

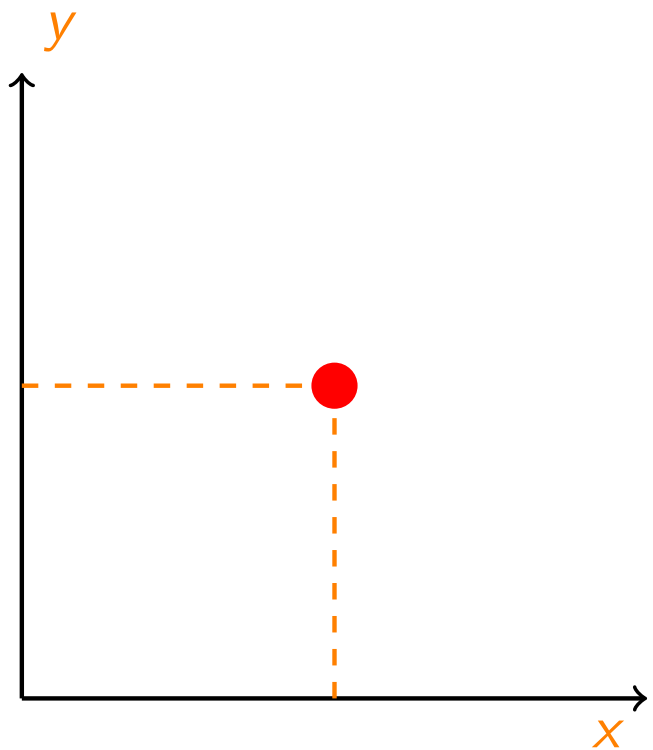


Figura: Um gráfico com os eixos X e Y e um ponto circular vermelho nas coordenadas X e Y

TAD – Exemplo 03

Ponto 2D

Vamos (re)definir o *ponto 2D* por meio de uma TAD...

(1) definir o arquivo `tad.h`:
   - protótipos das funções;
   - tipos de ponteiros;
   - dados globalmente acessíveis.

(2) definir o arquivo `tad.c`

(3) na condição de cliente, usar `main.c`

TAD – Exemplo 03

```
1  // Arquivo: ponto.h
2
3  typedef struct ponto Ponto;
4
5  Ponto* ponto_cria(float x, float y, bool visibilidade);
6
7  void ponto_libera(Ponto* p);
8
9  void ponto_acessa(Ponto* p, float* x, float* y);
10
11 void ponto_atribui(Ponto* p, float x, float y);
12
13 float ponto_distancia(Ponto* p1, Ponto* p2);
14
15 void ponto_oculta(Ponto* p);
16
17 void ponto_mostra(Ponto* p);
18
19 void ponto_move(Ponto* p, float x, float y);
```

TAD – Exemplo 03

```
1  // Arquivo ponto.c
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <math.h>
6  #include <stdbool.h>
7  #include "ponto.h" // inclusao do arquivo de cabecalho de usuÃ¡rio!
8
9  struct ponto {
10     float x;
11     float y;
12     bool visibilidade; // true = visivel, false = invisivel
13 };
14 //
15 // Aqui estara o codigo–fonte das implementacoes das rotinas.
```

TAD – Exemplo 03

```
1  //
2  // Cria um ponto
3  //
4  Ponto* ponto_cria (float x, float y, bool visibilidade) {
5          Ponto* p = (Ponto*) malloc(sizeof(Ponto));
6          if (p != NULL) {
7                  p->x = x;
8                  p->y = y;
9                  p->visibilidade = visibilidade;
10         }
11         return p;
12 }
```

TAD – Exemplo 03

```
1 //
2 // Libera (desaloca) um ponto...
3 //
4 void ponto_libera (Ponto* p) {
5     if (p != NULL) {
6         free(p);
7     }
8 }
```

TAD – Exemplo 03

```
1 //
2 // Acessa um ponto, coletando suas coordenadas
3 //
4 void ponto_acessa (Ponto* p, float* x, float* y) {
5     if (p != NULL) {
6             *x=p->x;
7             *y=p->y;
8         }
9 }
```

TAD – Exemplo 03

```
1 //
2 // Atribui coordenadas a um ponto, modificando–o
3 //
4 void ponto_atribui (Ponto* p, float x, float y) {
5     if (p != NULL) {
6         p->x=x;
7         p->y=y;
8     }
9 }
```

TAD – Exemplo 03

```
1  //
2  // Retorna a distancia entre dois pontos
3  //
4  float ponto_distancia (Ponto* p1, Ponto* p2) {
5          float dx, dy;
6
7          dx = p1->x - p2->x;
8          dy = p1->y - p2->y;
9          return (sqrt(dx*dx+dy*dy));
10 }
```

TAD – Exemplo 03

```
1  //
2  // move o ponto para as coordenadas (x, y)
3  //
4  void ponto_move (Ponto* p, float x, float y, int movimento) {
5
6        //
7        // Move o ponto p, a partir de sua posicao atual, para
8        // a posicao (x, y) segundo um certo tipo de movimento.
9        //
10       // Por exemplo: 1 — linear
11       // 2 — zig—zag
12       // ...
13       //
14
15 }
```

TAD – Exemplo 03

```
1 //
2 // Oculta (torna invisivel) o ponto
3 //
4 void ponto_oculta (Ponto* p) {
5
6     p->visibilidade = false;
7 }
```

TAD – Exemplo 03

```
1 //
2 // Mostra (torna visivel) o ponto
3 //
4 void ponto_mostra (Ponto* p) {
5
6     p->visibilidade = true;
7 }
```

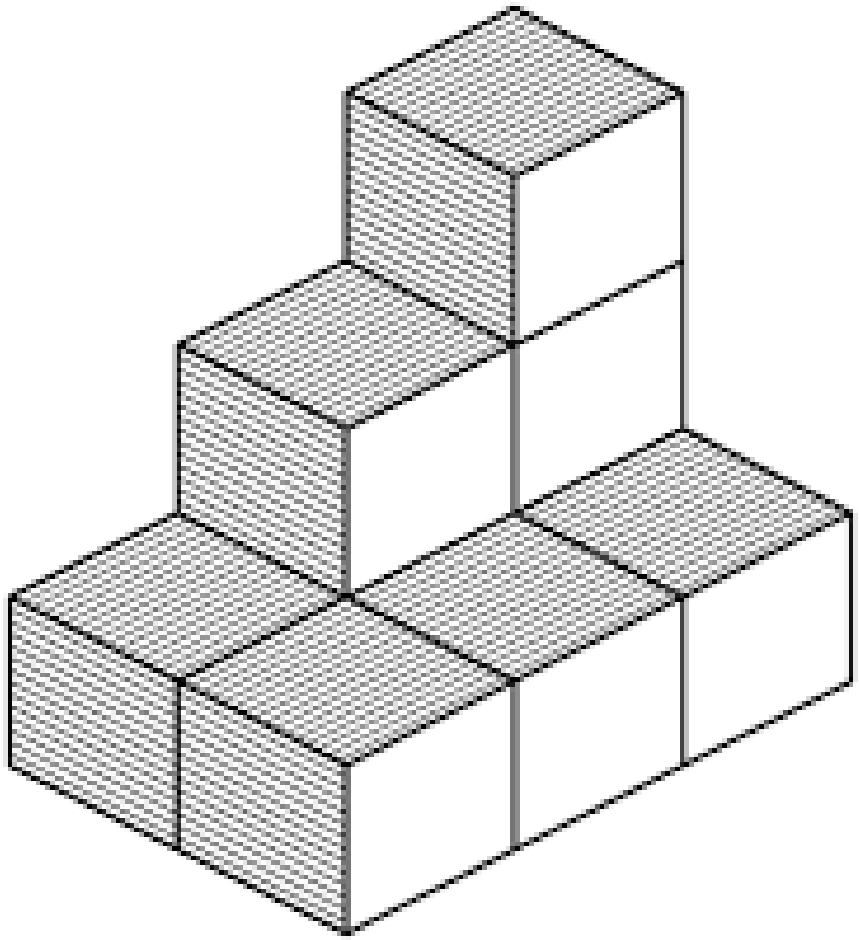
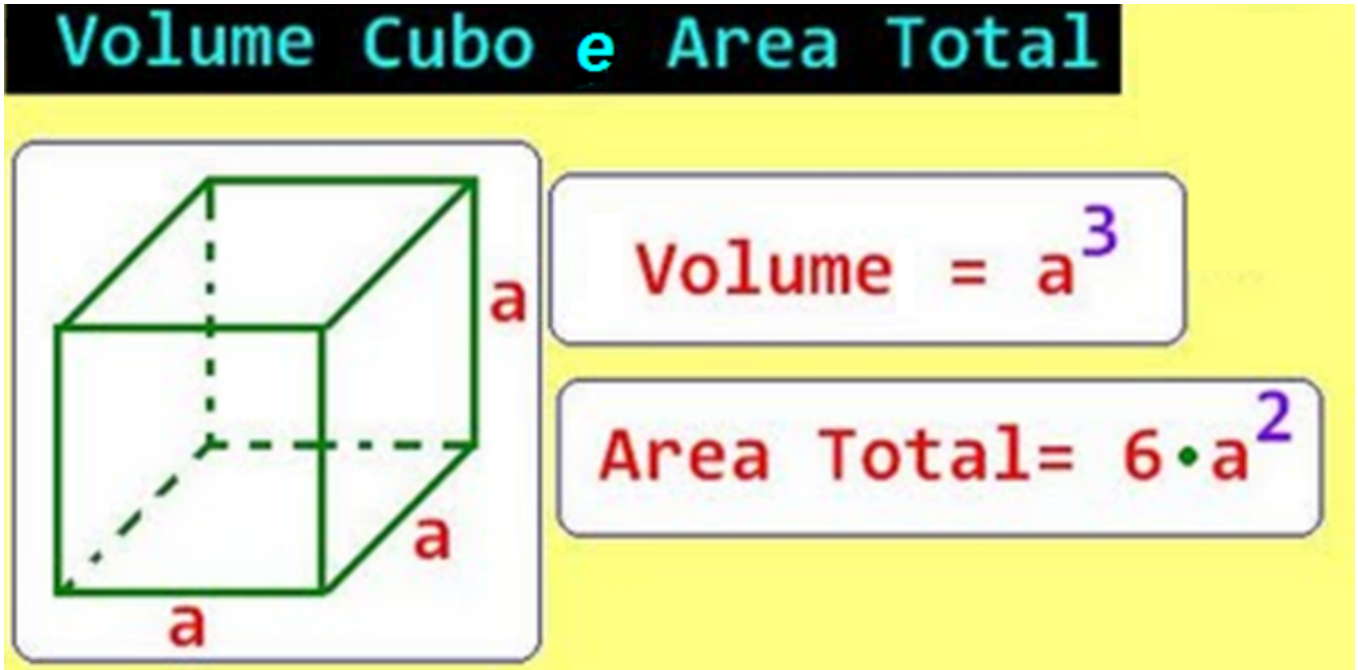


Figura: Imagem de vários cubos empiilhados

TAD – Exercícios

1 Conceba, planeje e implemente, em $\mathbb{C}$, um TAD para representar um cubo tridimensional.
São necessárias as seguintes operações, consideradas *fundamentais*:

(1) criar o cubo;
(2) destruir o cubo;
(3) retornar o comprimento da aresta;
(4) retornar o perímetro das arestas;
(5) retornar a área de uma face do cubo;
(6) retornar a área total das faces do cubo;
(7) retornar o volume do cubo;
(8) retornar o comprimento de suas diagonais.


Volume Cubo e Area Total
a
a
a
Volume = a^3
Area Total= $6 \cdot a^2$

Cubo.h

```
1  //
2  // cubo.h
3  //
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <math.h>
7
8  typedef struct cubo Cubo;
9
10 Cubo* cubo_cria(float a);
11 void cubo_libera(Cubo* c);
12 float cubo_acessa(Cubo* c);
13 void cubo_atribui(Cubo* c, float a);
14 float cubo_perimetro(Cubo* c);
15 float cubo_areaFace(Cubo* c);
16 float cubo_areaTotal(Cubo* c);
17 float cubo_volume(Cubo* c);
18 float cubo_diagonal(Cubo* c);
```

Cubo.c

```
1  //
2  // cubo.c
3  //
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <math.h>
7  #include "cubo.h"
8
9  struct cubo {
10     float a;
11 };
```

Cubo.c

```
1  //
2  // Cria um cubo
3  //
4  Cubo* cubo_cria (float a) {
5          Cubo* c = (Cubo*) malloc(sizeof(Cubo));
6          if (c != NULL) {
7                  c->a = a;
8          }
9          return (c);
10 }
```

Cubo.c

```
1 //
2 // Libera (desaloca) o cubo
3 //
4 void cubo_libera (Cubo* c) {
5       if (c != NULL) {
6             free(c);
7           }
8 }
```

Cubo.c

```
1 //
2 // Acessa um cubo
3 //
4 float cubo_acessa (Cubo* c) {
5         return (c->a);
6 }
```

Cubo.c

```
1 //
2 // Atribui um valor a aresta do cubo
3 //
4 void cubo_atribui (Cubo* c, float a) {
5         if (c != NULL) {
6             c->a = a;
7         }
8 }
```

Cubo.c

```
1 //
2 // Retorna o perimetro do cubo
3 //
4 float cubo_perimetro (Cubo* c) {
5         if (c != NULL) {
6             return(12 * c->a);
7         }
8 }
```

Cubo.c

```
1 //
2 // Retorna a area da face do cubo
3 //
4 float cubo_areaFace (Cubo* c) {
5         if (c != NULL) {
6             return(c->a * c->a);
7         }
8 }
```

Cubo.c

```
1 //
2 // Retorna a area total das faces do cubo
3 //
4 float cubo_areaTotal (Cubo* c) {
5         if (c != NULL) {
6             return(6 * c->a * c->a);
7         }
8 }
```

Cubo.c

```
1 //
2 // Retorna o volume do cubo
3 //
4 float cubo_volume (Cubo* c) {
5         if (c != NULL) {
6             return(c->a * c->a * c->a);
7         }
8 }
```

Cubo.c

```
1 //
2 // Retorna a diagonal do cubo
3 //
4 float cubo_diagonal (Cubo* c) {
5         if (c != NULL) {
6             return(sqrt(3) * c->a);
7         }
8 }
```

MainCubo.c

```c
//
// Corpo principal
//
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cubo.h"

int main(){
        float aresta;
        Cubo* variavelCubo;

        printf("digite o valor da aresta do cubo: ");
        scanf("%f",&aresta);
        variavelCubo = cubo_cria(aresta);
        printf("aresta = %.2f\n", cubo_acessa(variavelCubo));
        printf("perimetro = %.2f\n", cubo_perimetro(variavelCubo));
        printf("diagonal = %.2f\n", cubo_diagonal(variavelCubo));
        printf("area = %.2f\n", cubo_area(variavelCubo));
        printf("volume = %.2f\n", cubo_volume(variavelCubo));
        cubo_libera(variavelCubo);
        return 0;
}
```

Referências Bibliográficas

(1) ASCÊNCIO, A. F. G. e ARAÚJO, G. S. de. *Estruturas de dados*, São Paulo: Prentice Hall, 2010;

(2) CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C. *Introduction to algorithms*, 3ª edição, MIT Press, 2009;

(3) FERRARI, R.; RIBEIRO, M. X.; DIAS, R. L. e FALVO, M. *Estruturas de dados com jogos*, 1ª edição, São Paulo: Elsevier, 2014;

(4) GOODRICH, M. T. & TAMASSIA, R. *Estruturas de dados e algoritmos em Java*, 4ª edição, São Paulo: Bookman, 2007;

Referências Bibliográficas

(5) SEBESTA, Robert W., *Concepts in programming languages*, 10th. ed., Pearson, 2012;

(6) SKIENA, S. S. *The algorithm design manual*, 3^{th} ed., London:Springer-Verlag, 2020.

(7) TANENBAUM, A. A., LANGSAM, Y., e AUGUSTEIN, M. J. *Estruturas de dados usando C*, Makron Books, 1995;

(8) ZIVIANI, N. *Projeto de algoritmos: com implementações em Pascal e C*, 2 edição, Cengage Learning, 2009.

Saiba Mais...

...

- vídeos indicados na área da disciplina na Plataforma Turing do INF/UFG;
- vídeos disponíveis na *web*. Por exemplo, no YouTube;
- textos explicativos, e códigos-fonte, publicados em diversos *websites* que abordam estruturas de dados.