#### Tipos Abstratos de Dados

**Conceitos básicos**

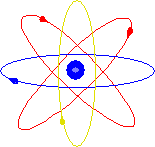
Tipo de Dados: conjunto de valores que uma variável pode assumir.

Tipos Primitivos de Dados: são aqueles pré-definidos na linguagem, a partir dos quais podemos definir outros tipos. Variam de linguagem para linguagem.

Tipos Compostos ou Estruturados de Dados: Estruturas de dados definidas a partir dos tipos primitivos de dados.

Estrutura de Dados: é um conjunto de variáveis, possivelmente de diferentes tipos de dados, conectadas de diferentes formas e que implementa uma abstração do mundo real (cria um **modelo** de uma porção de interesse do universo).

Níveis de agregação de Estruturas de Dados:

* Estruturas de dados homogêneas: ***vetores***
* Estruturas de dados heterogêneas: ***registros***

Tipo Abstrato de Dados - TAD: é um ***modelo matemático*** (implementado por uma ***Estrutura de Dados***) acompanhado de ***operações*** (implementadas por **funções**) definidas sobre este modelo. Representa uma **abstração** da realidade.

Exemplos de TAD:

* o conjunto de números inteiros acompanhado das operações de adição, subtração, multiplicação, etc.
* um ponto no plano cartesiano acompanhado das operações de criar um ponto, destruir um ponto, atribuir valores, medir distância entre dois pontos, etc.

Principais características de TAD:

* **Encapsulamento**

A idéia aqui é “esconder” do usuário a forma concreta como o modelo foi implementado.

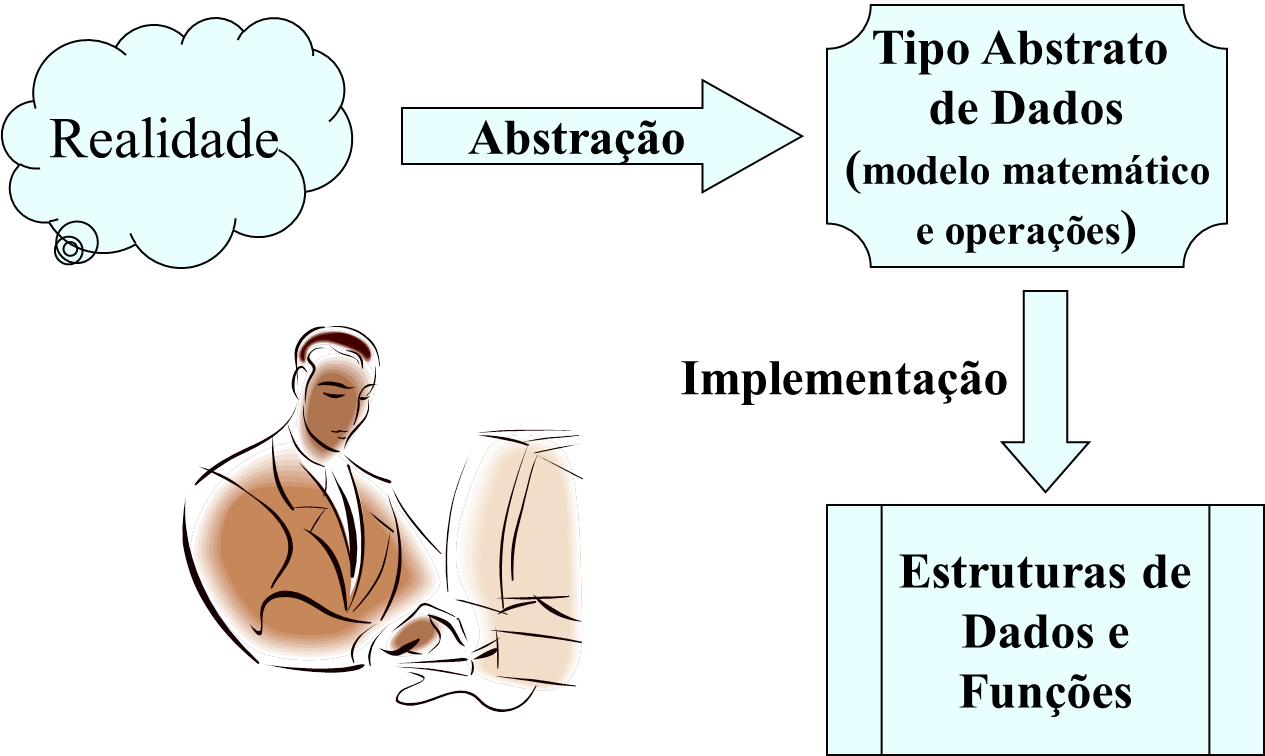
Por exemplo, no TAD números inteiros, usuários “enxergam” apenas as operações definidas (adição, subtração, multiplicação).

No TAD ponto, o usuário manipula pontos apenas por meio das operações Criar, Atribuir, Medir Distância...

* **Generalização**

A idéia aqui é aplicar os mesmos conceitos em diferentes situações ou, em outras palavras, “reutilizar” o mesmo modelo matemático e operações para manipular diferentes objetos do mundo real.

Os conceitos apresentados acima podem ser resumidos na figura a seguir:



**Exemplo prático de modelagem aplicando abstração (conjunto de números inteiros):**

**Realidade:**

**Necessidade de contar as nossas posses ( p. e., a quantidade de cabeças no meu rebanho)**

**Tipo Abstrato de Dados:**

**Conjunto de números inteiros e operações aritméticas**

**Z = { . . . , -3, -2, -1, 0, 1, 2, 3, . . . }**

**Op. = { + , - , 🞩 ,÷ }**

**Implementação**

**(** Na linguagem C **)**

**Estrutura de dados: int, short, long, . . .**

**Operações + , - , \* , / , % , . . .**

Note que este exemplo constitui os próprios *tipos primitivos* da linguagem C, ou seja, os tipos *int*, *short*, *long*, etc., com suas respectivas operações aritméticas constituem diferentes implementações do TAD Números Inteiros na linguagem C.

**Demo: implementação do TAD Ponto**

Já foi visto acima um exemplo demonstrando um Tipo Abstrato de Dados para representar um ponto no plano Cartesiano. Agora será demonstrada a implementação, na linguagem C, do TAD Ponto, que constitui as estruturas de dados e operações básicas sobre o ponto. Um ponto é genericamente representado por sua coordenada (abscissa X e ordenada Y). Adaptado de **CELES, Waldemar et ali**. Introdução a Estruturas de Dados, Rio de Janeiro: Campus, 2004, pag. 127.

Para implementar a estrutura de dados basta definir um registro com as coordenadas do ponto. Em Português Estruturado:

TipoPonto: **Registro**

x : real

y : real

**FimRegistro**

Em Linguagem C:

typedef struct ponto {

float x;

float y;

} Ponto;

As operações consideradas neste DEMO para o TAD Ponto estão descritas a seguir e implementadas respectivamente em Português Estruturado e na Linguagem C:

* **Criar Ponto**: Cria um ponto no plano cartesiano e inicializa seus campos com os valores da abscissa (x) e ordenada (y), recebidos como parâmetro.

**Funcao** criaPonto (x, y: real): TipoPonto

**Var** p TipoPonto

**Inicio**

*Alocar memória para* p

p.x 🡨 x;

p.y 🡨 y,

Retorne p

**FimFuncao**

Ponto\* criaPonto(float x, float y)

{

Ponto\* p = (Ponto\*) malloc(sizeof(Ponto));

if (p != NULL) { // p == null significa *overflow* de memória

p->x = x;

p->y = y;

}

return p;

}

* **Verificar Ponto**: operação *booleana* que verifica se um ponto já existe.

**Funcao** existePonto (p: TipoPonto): logico

**Inicio**

Retorne (p ≠ nulo)

**FimFuncao**

int existePonto(Ponto\* p)

{

return(p != NULL);

}

* **Destruir Ponto**: Deleta um ponto anteriormente criado e libera a memória alocada para este.

**Funcao** destroiPonto (p: TipoPonto): void

**Inicio**

*Liberar memória de* p

**FimFuncao**

void destroiPonto(Ponto\*\* p)

{

free(\*p);

\*p = NULL;

}

* **Iniciar Ponto**: Atribui um valor à abscissa (x) e ordenada (y) de um dado ponto p.

**Funcao** atribuiPonto (x, y: real): TipoPonto

**Var** p TipoPonto

**Inicio**

p.x 🡨 x;

p.y 🡨 y,

Retorne p

**FimFuncao**

void atribuiPonto(Ponto\* p, float x, float y)

{

p->x = x;

p->y = y;

}

* **Acessar Ponto**: Recupera os valores da abscissa (x) e ordenada (y) de um dado ponto p.

**Funcao** acessaPonto (**var** p TipoPonto, **var** x, y: real): void

**Inicio**

x 🡨 p.x;

y 🡨 p.y,

**FimFuncao**

void acessaPonto(Ponto\* p, float\* x, float\* y)

{

\*x = p->x;

\*y = p->y;

}

* **Calcular Distância**: calcula a distância entre os pontos p1 e p2, fornecidos como parâmetros.

**Funcao** distanciaPontos (p1, p2: TipoPonto): real

**Var** dx, dy: real

**Inicio**

dx 🡨 p2.x – p1.x;

dy 🡨 p2.y – p1.y;

Retorne (dx\*dx + dy\*dy)\*\*(1/2)

**FimFuncao**

float distanciaPontos(Ponto\* p1, Ponto\* p2)

{

float dx = p2->x - p1->x;

float dy = p2->y - p1->y;

return sqrt(dx\*dx + dy\*dy);

}

Observe que o TAD Ponto implementado acima pode ser utilizado por um aplicativo para manipular pontos abstraídos em um plano cartesiano. Mas o TAD Ponto, por si só, não é executável de forma independente.

**Aplicação Prática – TAD Círculo**

Um círculo pode ser abstraído como um centro (um ponto) e um raio (distância do centro ao perímetro do círculo). Implemente um **TAD Círculo**, em Português Estruturado ou Linguagem C. Deve ser implementada a estrutura do círculo (note que esta poderá reutilizar o **TAD Ponto**, já implementado!) e as seguintes operações:

* **Criar Círculo**: recebe como parâmetro um ponto e um raio, cria um novo círculo e inicia este com os parâmetros recebidos.
* **Verificar Círculo**: operação *booleana* que verifica se um círculo já existe.
* **Destruir Círculo**: deleta um círculo anteriormente criado e libera a memória alocada para este.
* **Area do Círculo**: calcula e retorna a área do circulo de raio R – fórmula: Area = Pi \* R2.
* **Pertence ao Círculo**: verifica se um ponto pertence ao circulo. Um ponto pertence ao círculo se a distância do ponto ao centro for igual ao Raio (Use operação *distanciaPontos* do TAD Ponto!).
* **Interno ao Círculo**: verifica se um ponto é interno ao circulo. Um ponto é interno se a distância do ponto ao centro for menor que Raio.

**Solução para o TAD Círculo**: considerando a existencia do **TAD Ponto**, definido acima.

Em Português Estruturado

TipoCirculo: **Registro**

centro : TipoPonto

raio : real

**FimRegistro**

**Funcao** criaCirculo (c: TipoPonto, r: real): TipoCirculo

**Var** circulo: TipoCirculo

**Inicio**

*Alocar memória para* circulo

circulo.centro 🡨 c

circulo.raio 🡨 r

Retorne circulo

**FimFuncao**

**Funcao** existeCirculo (c: TipoCirculo): logico

**Inicio**

Retorne (c ≠ nulo)

**FimFuncao**

**Funcao** destroiCirculo (c: TipoCirculo): void

**Inicio**

*Liberar memória de* c

**FimFuncao**

**Funcao** areaCirculo (r: real): real

**Const** PI = 3.14159

**Var** area: real

**Inicio**

area 🡨 PI \* r \*\* 2

Retorne area

**FimFuncao**

**Funcao** pertenceCirculo (c: TipoCirculo, p: TipoPonto): logico

**Var** dist: real

**Inicio**

dist 🡨 distanciaPontos(c.centro, p)

Retorne (dist == c.raio)

**FimFuncao**

**Funcao** internoCirculo (c: TipoCirculo, p: TipoPonto): logico

**Var** dist: real

**Inicio**

dist 🡨 distanciaPontos(c.centro, p)

Retorne (dist < c.raio)

**FimFuncao**