@RESUMOSCC

Tipo Abstrato de Dados

Resumo completo com exemplos em C — Estrutura de Dados

01

O que é um Tipo de Dado?

Um tipo de dado define qual informação pode ser armazenada em uma variável e qual valor é retornado por uma função. Matematicamente, é formado por um conjunto de operandos e por um conjunto de operadores que transformam esses operandos.

Existem três categorias principais de tipos de dados:

Primitivos (ou simples): não são definidos em termos de outros tipos. Em C temos int, char, double, float. São a base da linguagem — como os 'átomos' da programação.

Estruturados: agrupam tipos primitivos em um único tipo. Em C, o struct permite criar tipos estruturados. Um vetor de char (char*) também é um tipo estruturado.

Abstratos (TAD): definidos pelo programador, com dados e operações encapsulados. A implementação fica oculta do usuário.

02

Definição de TAD

O Tipo Abstrato de Dado (TAD) é a forma elegante de definirmos um novo tipo de dado. Um TAD é um modelo matemático que possui:

Um **conjunto de valores** (os operandos)

Um **conjunto de operações** que atuam sobre esses valores (os operadores)

A grande diferença em relação aos outros tipos é que a definição de um TAD não está vinculada a linguagens de programação nem a implementações específicas. Por isso o termo abstrato: o TAD define O QUE o tipo faz, e não COMO ele faz internamente.

03

TAD vs Estrutura de Dados

É comum confundir os dois conceitos. A distinção é importante:

Estrutura de Dados: define a organização concreta dos dados na memória do computador. É uma implementação real. Por exemplo, um vetor organiza inteiros em sequência na memória.

TAD: é abstrato — define os dados e as operações sem se preocupar com a implementação. Um TAD se concretiza utilizando uma estrutura de dados para armazenar os valores e implementando as funções que manipulam essa estrutura.

04

Encapsulamento e Interfaces

Para transformar um TAD em código, usamos **interfaces**. Uma interface define a entrada e a saída de uma função sem revelar como ela funciona internamente.

Em C, a separação entre arquivos de cabeçalho (.h) e de código (.c) é a forma padrão de apresentar interfaces ao usuário. Ao incluir o <stdio.h>, por exemplo, você sabe O QUE as funções fazem e quais são seus parâmetros — sem precisar saber COMO elas funcionam internamente.

As operações típicas de um TAD encapsulado são: **Criar, Destruir, Atualizar e Consultar**.

05

Exemplo: TAD Ponto2D

Imagine que queremos representar um ponto no plano cartesiano (x, y) e calcular a distância entre dois pontos. Uma implementação ingênua usaria variáveis soltas — o que não constitui um TAD, pois não há conjunto de valores bem definido nem operações específicas sobre eles.

A solução correta é criar um TAD Ponto2D com:

Dois números decimais representando x e y

Operações de criação, recuperação, modificação e cálculo de distância

Um mesmo TAD pode ter implementações diferentes — o que importa é que todas cumpram o contrato definido.

06

Exemplo: TAD Relógio

Um TAD Relógio representa um horário (horas, minutos e segundos) com operações para criar, verificar e modificar a hora. A implementação pode variar, mas a interface permanece a mesma.

07

Observação Final

Em C, usamos cabeçalhos (.h) para representar TADs — mas essa abordagem tem limitações, pois não permite esconder completamente os dados internos do usuário. Linguagens como C++, Java e Python possuem o conceito de **classes**, que são mais precisas para representar TADs, permitindo definir dados privados (não visíveis ao usuário). Você verá isso em outras disciplinas.

REFERÊNCIA DE CÓDIGO

Todos os códigos dos exemplos

Exemplos completos em C extraídos dos slides da aula

1. Ponto 2D sem TAD (implementação ingênua)

main.c — variáveis soltas, sem encapsulamento:

```
int main(){
  float x1 = 10;
  float y1 = 20;
  float x2 = 15;
  float y2 = 30;

  printf("Ponto 1 = (%f,%f)\n", x1, y1);
  printf("Ponto 2 = (%f,%f)\n", x2, y2);
  printf("D = %lf\n", distance(x1, y1, x2, y2));
  return 0;
}
```

```
double distance(float x1, float y1, float x2, float y2){
  double x_pow = pow((x1 - x2), 2);
  double y_pow = pow((y1 - y2), 2);
  return sqrt(x_pow + y_pow);
}
```

2. TAD Ponto2D — Interface (ponto2d.h)

O usuário só conhece o tipo e as assinaturas das funções:

```
typedef struct coord Ponto2D;

Ponto2D* cria_ponto2d(float, float);
void recupera(Ponto2D*, float*, float*);
void modifica(Ponto2D*, float, float);
double distancia(Ponto2D*, Ponto2D*);
```

3. TAD Ponto2D — Implementação com float x, y (ponto2d.c)

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

typedef struct coord Ponto2D;

Ponto2D* cria_ponto2d(float x, float y){
  Ponto2D *c = (Ponto2D*)malloc(sizeof(Ponto2D));
  c->x = x;
```

```
  c->y = y;
  return c;
}

void recupera(Ponto2D* p, float* x, float* y){ ... }
void modifica(Ponto2D* p, float x, float y){ ... }
double distancia(Ponto2D* p1, Ponto2D* p2){ ... }
```

4. TAD Ponto2D — Implementação alternativa com vetor

Mesma interface, implementacao interna diferente:

```
struct coord{
  float pontos[2];
};

typedef struct coord Ponto2D;

Ponto2D* cria_ponto2d(float x, float y){
  Ponto2D *c = (Ponto2D*)malloc(sizeof(Ponto2D));
  c->pontos[0] = x;
  c->pontos[1] = y;
  return c;
}
```

5. Usando o TAD Ponto2D no main

```
int main(){
  Ponto2D* p1 = cria_ponto2d(10, 20);
  Ponto2D* p2 = cria_ponto2d(15, 30);

  printf("D = %lf\n", distancia(p1, p2));
  return 0;
}
```

6. TAD Relógio — Interface (relogio.h)

```
typedef struct relogio Relogio;

Relogio* cria_relogio();
void verifica_hora(Relogio*, int*, int*, int*);
void modifica_hora(Relogio*, int, int, int);
```

7. TAD Relógio — Implementação (relogio.c)

```
struct relogio{
  int H;
  int M;
  int S;
};
```

```c
typedef struct relogio Relogio;

Relogio* cria_relogio(){
  Relogio *r = (Relogio*)malloc(sizeof(Relogio));
  r->H = 0;
  r->M = 0;
  r->S = 0;
  return r;
}

void verifica_hora(Relogio* r, int* h, int* m, int* s){
  *h = r->H;
  *m = r->M;
  *s = r->S;
}

void modifica_hora(Relogio* r, int h, int m, int s){
  r->H = h;
  r->M = m;
  r->S = s;
}
```