


# Tipo abstrato de dados – TAD


## 4.1 DEFINIÇÃO

Quando iniciamos nossos estudos em uma linguagem de programação, vários conceitos foram apresentados. Um desses conceitos foi o de **tipo de dados**.



Um **tipo de dado** define o conjunto de **valores** (domínio) e **operações** que uma variável pode assumir.


Por exemplo, o tipo **char**, da linguagem C, suporta valores inteiros que vão de -128 até +127. Além disso, esse tipo também suporta operações de soma, subtração etc. Vimos também que não possuem nenhum tipo de estrutura sobre seus valores. Porém, existem outros tipos de dados, chamados **dados estruturados** ou **estruturas de dados**. Neles, existe uma relação estrutural (que pode ser **linear** ou **não linear**) entre seus valores.



Uma **estrutura de dados** consiste em um conjunto de tipos de dados em que existe algum tipo de relacionamento lógico estrutural.


Ou seja, uma **estrutura de dados** é apenas uma forma de armazenar e organizar os dados de modo que eles possam ser usados de forma eficiente. Alguns exemplos das estruturas de dados presentes na linguagem C são os **array**, **struct**, **union** e **enum**, todas criadas a partir dos tipos de dados básicos.

Às vezes, os tipos de dados e as estruturas de dados presentes na linguagem podem não ser suficientes para nossa aplicação. Podemos necessitar de uma melhor estruturação dos dados, assim como precisamos especificar quais operações estarão disponíveis para manipular esses dados. Neste caso, convém criar um **tipo abstrato de dados**, também conhecido como **TAD**.



Um **tipo abstrato de dados**, ou **TAD**, é um conjunto de dados estruturados e as operações que podem ser executadas sobre esses dados.

Basicamente, o **tipo abstrato de dados** é um conjunto de valores com seu comportamento definido por operações implementadas na forma de funções. Ele é construído a partir dos tipos básicos (**int**, **char**, **float** e **double**) ou dos tipos estruturados (**array**, **struct**, **union** e **enum**) da linguagem C. Assim, **tipos abstratos de dados** são entidades puramente teóricas, usadas para simplificar a descrição de algoritmos abstratos, classificar e avaliar as estruturas de dados e descrever formalmente certos tipos de sistemas.



Tanto a representação quanto as operações do **TAD** são especificadas pelo programador. O usuário utiliza o **TAD** como uma caixa-preta por meio de sua **interface**.

Para a criação de um **TAD** é essencial ocultar os dados do usuário, ou seja, devemos tornar invisível a sua implementação para o usuário. Assim, o **TAD** é como uma caixa-preta para o usuário, que nunca tem acesso direto à informação lá armazenada. A implementação de um **TAD** está desvinculada da sua utilização, ou seja, quando definimos um **TAD**, estamos preocupados com o que ele faz e não em como ele faz.

Um **TAD** é, muitas vezes, implementado na forma de dois módulos: **implementação** e **interface**. O módulo de **interface** declara as funções que correspondem às operações do **TAD** e é visível pelo usuário. A estratégia de ocultação de informações permite a implementação e a manutenção de módulos sem afetar os programas do usuário, como mostra a Figura 4.1.

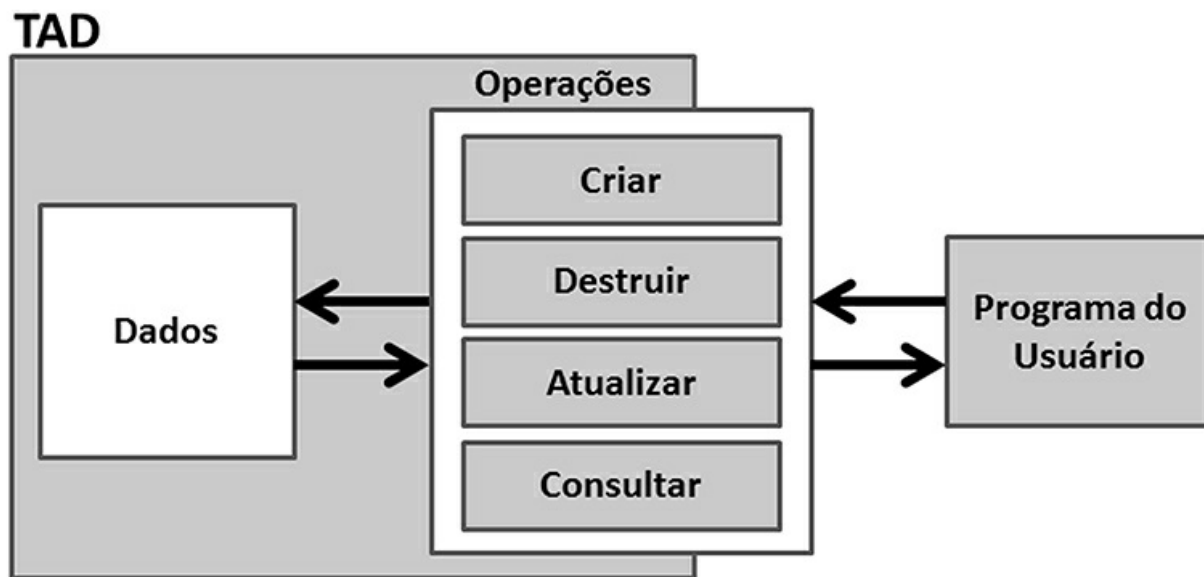


FIGURA 4.1

#### 4.1.1 Vantagens de usar um TAD

O uso de um **TAD** traz consigo uma série de vantagens:

- **Encapsulamento:** ao ocultarmos a implementação, fornecemos um conjunto de operações possíveis para o **TAD**. Isso é tudo o que o usuário precisa saber para fazer uso do **TAD**. O usuário não precisa de nenhum conhecimento técnico sobre como a implementação trabalha para usá-lo, tornando o seu uso muito mais fácil.
- **Segurança:** o usuário não tem acesso direto aos dados. Isso evita que ele manipule os dados de uma maneira imprópria.
- **Flexibilidade:** podemos alterar o **TAD** sem alterar as aplicações que o utilizam. De fato, podemos ter diferentes implementações de um **TAD**, desde que todos respeitem a mesma interface. Assim, podemos usar a implementação mais eficiente para determinada situação.
- **Reutilização:** a implementação do **TAD** é feita em um módulo diferente do programa do usuário.

#### 4.1.2 O tipo FILE

Se você já trabalhou com arquivos na linguagem C, então muito provavelmente já teve o seu primeiro contato com um tipo abstrato de dados, **TAD**. Trata-se do tipo **FILE**.

O tipo **FILE** é uma estrutura que contém as informações sobre um arquivo ou fluxo de texto necessário para realizar as operações de entrada ou saída sobre ele, tais como o descritor do arquivo, a posição atual dentro do arquivo, um indicador de fim de arquivo e um indicador de erro, como mostra a Figura 4.2. Tenha em mente que o conteúdo dessa estrutura parece mudar significativamente em outras implementações.

## Exemplo: estrutura do tipo FILE

```
01 typedef struct{
02     int      level;          // nível do buffer
03     unsigned flags;          // flag de status do arquivo
04     char      fd;             // descritor do arquivo
05     unsigned char hold;      // retorna caractere se sem buffer
06     int       bsize;          // tamanho do Buffer
07     unsigned char *buffer;    // buffer de transferência de dados
08     unsigned char *curp;      // ponteiro atualmente ativo
09     unsigned   istemp;        // indicador de arquivo temporário
10     short      token;         // usado para validação
11 } FILE;
12
```

FIGURA 4.2

Alguns acreditam que ninguém, em sã consciência, deve fazer uso direto dos campos dessa estrutura. Então, a única maneira de trabalhar com arquivos em linguagem C é declarando um ponteiro de arquivo da seguinte maneira:

```
FILE* f;
```

Desse modo, o usuário possui apenas um ponteiro para onde os dados estão armazenados, mas não pode acessá-los diretamente.



A única maneira de acessar o conteúdo do ponteiro **FILE** é por meio das operações definidas em sua **interface**.

Assim, os dados do ponteiro **f** somente podem ser acessados pelas funções de manipulação do tipo **FILE**:

- `fopen()`
- `fclose()`
- `fputc()`
- `fgetc()`
- `feof()`
- etc

### 4.1.3 Tipo opaco

Sempre que trabalhamos com arquivos na linguagem C temos a necessidade de declarar um ponteiro do tipo **FILE** para poder manipular o arquivo.



Se o tipo **FILE** é, na verdade, uma estrutura, por que não podemos simplesmente declarar uma variável em vez de um ponteiro para ela?

Isso acontece porque o tipo **FILE** é um tipo **opaco** do ponto de vista dos usuários da biblioteca. Apenas a própria biblioteca conhece o conteúdo do tipo e consegue manipulá-lo.



Diz-se que um tipo de dado é opaco quando ele é incompletamente definido em uma interface. Assim, os seus valores só podem ser acessados por funções específicas.

Basicamente, um tipo opaco representa uma forma de esconder os detalhes de sua implementação dos programadores que apenas farão uso do módulo ou biblioteca. Para criar um tipo opaco, utilizamos dois arquivos e os princípios de modularização:

- Arquivo “.C”: declara o tipo de dados que deverá ficar oculto do usuário.
- Arquivo “.H”: declara o tipo que irá representar os dados ocultos do arquivo “.C” e que somente poderá ser declarado pelo usuário na forma de um ponteiro.

Por meio dos tipos opacos, nossos programas utilizam uma biblioteca apenas através de ponteiros para os dados. E os dados somente podem ser acessados por funções. Note que é justamente isso que acontece quando trabalhamos com arquivos na linguagem C.

#### 4.1.4 Operações básicas de um TAD


Tipos abstratos de dados incluem as operações para a manipulação de seus dados. Essas operações variam de acordo com o **TAD** criado, porém as seguintes operações básicas são possíveis:

- Criação do **TAD**.
- Inserção de um novo elemento no **TAD**.
- Remoção de um elemento do **TAD**.
- Acesso a um elemento do **TAD**.
- Destruição do **TAD**.

## 4.2 MODULARIZANDO O PROGRAMA


Quando trabalhamos com **TAD**, a convenção em linguagem C é prepararmos dois arquivos para implementá-lo. Assim, podemos separar o “conceito” (definição do tipo) de sua “implementação”:

- Um arquivo “.H”: são declarados os protótipos das funções visíveis para o usuário, os tipos de ponteiro e os dados globalmente acessíveis. Aqui, é definida a **interface** visível pelo usuário.
- Um arquivo “.C”: declaração do tipo de dados que ficarão ocultos do usuário do **TAD** e implementação das suas funções. Aqui, é definido tudo que ficará **oculto** do usuário.



A esse processo de separação da definição do **TAD** em dois arquivos damos o nome de **modularização**.


Em programação, a **modularização** visa à criação de **módulos**. Um módulo é uma unidade com um propósito único e bem definido que pode ser compilado separadamente do restante do programa. Desse modo, um módulo pode ser facilmente reutilizado e modificado independentemente do programa do usuário.



Um módulo pode conter um ou mais tipos, variáveis, constantes e funções, além de uma interface apropriada com outros módulos existentes.

À medida que uma aplicação se torna maior, o uso de módulos se torna necessário. Isso ocorre porque o uso de um único arquivo causa uma série de problemas:

- Redação e modificação do programa se tornam mais difíceis, à medida que ele fica maior.
- A reutilização do código é um processo de copiar e colar.
- Qualquer modificação exige a recompilação de todo o código.
- O código é dividido em arquivos separados com entidades relacionadas.




A criação de módulos usa a estratégia de “dividir para conquistar” para a resolução de problemas.

A ideia básica da modularização é **dividir para conquistar**. Esta estratégia apresenta uma série de vantagens:

- Divide-se um problema maior em um conjunto de problemas menores.
- Aumenta as possibilidades de reutilização do código.
- Facilita o entendimento do programa.
- Encapsulam-se os dados. Assim, são agrupados os dados e processos logicamente relacionados.
- Código fica distribuído em vários arquivos. Isso permite trabalhar com equipes de programadores.
- Aumento da produtividade do programador.
- Apenas as partes alteradas do programa precisam ser recompiladas.
- Verificação independente dos módulos antes do uso no programa do usuário.
- Maior confiabilidade.


### 4.3 IMPLEMENTANDO UM TAD: PONTO

Nesta seção iremos mostrar passo a passo a criação de um **TAD**. Para o nosso exemplo, iremos criar um **TAD** que represente um ponto definido por suas coordenadas **x** e **y**. Como também estamos trabalhando com modularização, precisamos definir o **tipo opaco** que irá representar nosso ponto. Este tipo será um ponteiro para a estrutura que define o ponto. Além disso, também precisamos definir o conjunto de funções que serão visíveis para o programador que utilizar a biblioteca que estamos criando.

	No arquivo <b>Ponto.h</b> , iremos declarar tudo aquilo que será visível para o programador.
---	--

Vamos começar definindo o arquivo **Ponto.h**, ilustrado na Figura 4.3. Nele, temos que estabelecer:

- Um novo nome (**ponto**) para a **struct ponto** (linha 1). Esse é o tipo opaco que será usado sempre que se desejar trabalhar com nosso **TAD**.
- As funções disponíveis para trabalhar com nosso **TAD** (linhas 2-11) e que serão implementadas no arquivo **Ponto.c**.

	No arquivo <b>Ponto.c</b> iremos definir tudo aquilo que deve ficar oculto do usuário da nossa biblioteca e implementar as funções definidas em <b>Ponto.h</b> .
---	--

Basicamente, o arquivo **Ponto.c** (Figura 4.3) contém apenas:

- As chamadas às bibliotecas necessárias à implementação do **TAD** (linhas 1-3).
- A definição do tipo que descreve o nosso **TAD**, **struct ponto** (linhas 4-7).
- As implementações das funções definidas no arquivo **Ponto.h**, as quais serão vistas adiante.

Por estarem definidos dentro do arquivo **.c**, os campos da estrutura **struct ponto** não são visíveis pelo usuário da biblioteca no arquivo **main()**, apenas o seu outro nome, definido no arquivo **Ponto.h** (linha 1), que pode apenas declarar um ponteiro para ele, da seguinte forma:

Ponto \*p;

## Arquivo Ponto.h

```
01  typedef struct ponto Ponto;  
02  //Cria um novo ponto  
03  Ponto* Ponto_cria(float x, float y);  
04  //Libera um ponto  
05  void Ponto_libera(Ponto* p);  
06  //Acessa os valores "x" e "y" de um ponto  
07  int Ponto_acessa(Ponto* p, float* x, float* y);  
08  //Atribui os valores "x" e "y" a um ponto  
09  int Ponto_atribui(Ponto* p, float x, float y);  
10  //Calcula a distância entre dois pontos  
11  float Ponto_distancia(Ponto* p1, Ponto* p2);
```

## Arquivo Ponto.c

```
01  #include <stdlib.h>  
02  #include <math.h>  
03  #include "Ponto.h" //inclui os Protótipos  
04  struct ponto{//Definição do tipo de dados  
05      float x;  
06      float y;  
07  };
```

FIGURA 4.3


Para utilizar um **TAD** em seu programa, a primeira coisa a fazer é criar um novo ponto. Essa tarefa é executada pela função descrita na Figura 4.4. Basicamente, o que esta função faz é alocar uma área de memória para o **TAD** (linha 2). Esta área corresponde à memória necessária para armazenar a estrutura que define o ponto armazenado no **TAD**, **struct ponto**, a qual é devolvida para o nosso ponteiro para o **TAD**. Em seguida, essa função inicializa os campos da estrutura com os valores fornecidos pelo usuário (linhas 4-5) e retorna para o usuário o **TAD** (linha 7).

## Criando um ponto

```
01  Ponto* Ponto_cria(float x, float y){  
02      Ponto* p = (Ponto*) malloc(sizeof(Ponto));  
03      if(p != NULL){  
04          p->x = x;  
05          p->y = y;  
06      }  
07      return p;  
08  }
```

FIGURA 4.4

Sempre que terminarmos de utilizar nosso **TAD**, é necessário que ele seja destruído, como mostra o código contido na Figura 4.5. Basicamente, o que temos que fazer é liberar a memória alocada para a estrutura que representa o ponto. Isso é feito utilizando apenas uma chamada da função **free()**.



Por que criar uma função para destruir o **TAD** sendo que tudo que precisamos fazer é chamar a função **free()**?

Por questões de modularização. Destruir nosso **TAD Ponto** é uma tarefa simples, porém para outros **TADs** essa pode ser uma tarefa mais complicada. Ao criar essa função, estamos escondendo a implementação dessa tarefa do usuário, que não precisa saber como um **TAD** é destruído para utilizá-lo.

Destruindo um ponto

```
01 void Ponto_libera(Ponto* p) {
02     free(p);
03 }
```

FIGURA 4.5

Outra tarefa importante é o acesso à informação armazenada dentro do **TAD**. É preciso criar uma função que recupere, por referência, o valor das coordenadas de um ponto, como mostra o código contido na Figura 4.6. Primeiramente, a função verifica se o **TAD** é válido, isto é, se o ponteiro **Ponto\* p** é igual a **NULL**. Se essa condição for verdadeira, a função retorna o valor **ZERO** (linha 3), indicando erro na operação. Caso contrário, os dados são copiados para o conteúdo dos ponteiros passados por referência para a função (linhas 4 e 5) e a função retorna o valor **UM**, indicando sucesso nessa operação.

Acessando o conteúdo de um ponto

```
01 int Ponto_acessa(Ponto* p, float* x, float* y) {
02     if(p == NULL)
03         return 0;
04     *x = p->x;
05     *y = p->y;
06     return 1;
07 }
```

FIGURA 4.6

Podemos também querer modificar o valor atribuído ao **TAD**. Assim, é preciso criar uma função que atribua um novo valor às coordenadas de um ponto, como mostra o código contido na Figura 4.7. Primeiramente, a função verifica se o **TAD** é válido, isto é, se o ponteiro **Ponto\* p** é igual a **NULL**. Se essa condição for verdadeira, a função retorna o valor **ZERO** (linha 3), indicando erro na operação. Caso contrário, os dados passados por parâmetro são copiados para dentro da estrutura que representa o **TAD** (linhas 4-5) e a função retorna o valor **UM**, indicando sucesso nessa operação.



### Atribuindo um valor ao ponto

```
01  int Ponto_atribui(Ponto* p, float x, float y) {
02      if(p == NULL)
03          return 0;
04      p->x = x;
05      p->y = y;
06      return 1;
07  }
```

FIGURA 4.7

Dependendo de nossa aplicação, pode ser interessante saber a distância entre dois pontos. Então, vamos criar uma função que calcule a distância entre as coordenadas de dois **TAD ponto**, como mostra o código contido na Figura 4.8. Primeiramente, a função verifica se os dois **TADs** são válidos. Se algum deles for igual a **NULL**, a função irá retornar o valor **-1** (linha 3), indicando erro na operação (não existem distâncias negativas). Caso contrário, será calculada e retornada para o usuário da função o valor da distância entre os pontos (linhas 4-6).

### Calculando a distância entre dois pontos

```
01  float Ponto_distancia(Ponto* p1, Ponto* p2) {
02      if(p1 == NULL || p2 == NULL)
03          return -1;
04      float dx = p1->x - p2->x;
05      float dy = p1->y - p2->y;
06      return sqrt(dx * dx + dy * dy);
07  }
```

FIGURA 4.8

Por fim, podemos ver na Figura 4.9 um exemplo de como podemos utilizar nosso **TAD** em uma aplicação.

## Exemplo: utilizando o TAD ponto

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  #include "Ponto.h"
04  int main(){
05      float d;
06      Ponto *p,*q;
07      //Ponto r; //ERRO
08      p = Ponto_cria(10,21);
09      q = Ponto_cria(7,25);
10      //q->x = 2; //ERRO
11      d = Ponto_distancia(p,q);
12      printf("Distancia entre pontos: %f\n",d);
13      Ponto_libera(q);
14      Ponto_libera(p);
15      system("pause");
16      return 0;
17  }
```

FIGURA 4.9

## 4.4 EXERCÍCIOS

- 1) O que é um tipo abstrato de dados (TAD)? Qual a característica fundamental na sua utilização?
- 2) Quais as vantagens de se programar utilizando um TAD?
- 3) Desenvolva um TAD que represente um cubo. Inclua as funções de inicializações necessárias e as operações que retornem os tamanhos de cada lado, a sua área e o seu volume.
- 4) Desenvolva um TAD que represente um cilindro. Inclua as funções de inicializações necessárias e as operações que retornem a sua altura e o raio, a sua área e o seu volume.
- 5) Desenvolva um TAD que represente uma esfera. Inclua as funções de inicializações necessárias e as operações que retornem o seu raio, a sua área e o seu volume.
- 6) Desenvolva um TAD que represente um número complexo  $z = x + iy$ , em que  $i^2 = -1$ , sendo  $x$  a sua parte real e  $y$  a parte imaginária. O TAD deverá conter as seguintes funções:
  - Criar um número complexo.
  - Destruir um número complexo.
  - Soma de dois números complexos.
  - Subtração de dois números complexos.
  - Multiplicação de dois números complexos.
  - Divisão de dois números complexos.
- 7) Desenvolva um TAD que represente um conjunto de inteiros. Para isso, utilize um vetor de inteiros. O TAD deverá conter as seguintes funções:
  - Criar um conjunto vazio.
  - União de dois conjuntos.
  - Inserir um elemento no conjunto.
  - Remover um elemento do conjunto.
  - Intersecção entre dois conjuntos.
  - Diferença de dois conjuntos.
  - Testar se um número pertence ao conjunto.
  - Menor valor do conjunto.
  - Maior valor do conjunto.
  - Testar se dois conjuntos são iguais.
  - Tamanho do conjunto.
  - Testar se o conjunto é vazio.