

TAD - Tipos Abstratos de Dados

Estrutura de Dados — QXD0010



UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará

1º semestre/2021



Introdução



Tipos Abstratos de Dados (TADs)

- Um **Tipo Abstrato de Dado (TAD)** é uma especificação de um conjunto de dados e operações que podem ser executadas sobre esses dados.
 - Agrupa a estrutura de dados juntamente com as operações que podem ser feitas sobre esses dados

Tipos Abstratos de Dados (TADs)

- Um **Tipo Abstrato de Dado (TAD)** é uma especificação de um conjunto de dados e operações que podem ser executadas sobre esses dados.
 - Agrupa a estrutura de dados juntamente com as operações que podem ser feitas sobre esses dados
- A ideia central é **encapsular** (esconder) de quem usa um determinado tipo de dado a forma concreta com que ele foi implementado.
- Os usuários do TAD só têm acesso a algumas operações disponibilizadas sobre esses dados. Eles não têm acesso a detalhes de implementação.

Tipos Abstratos de Dados (TADs)

- Um **Tipo Abstrato de Dado (TAD)** é uma especificação de um conjunto de dados e operações que podem ser executadas sobre esses dados.
 - Agrupa a estrutura de dados juntamente com as operações que podem ser feitas sobre esses dados
- A ideia central é **encapsular** (esconder) de quem usa um determinado tipo de dado a forma concreta com que ele foi implementado.
- Os usuários do TAD só têm acesso a algumas operações disponibilizadas sobre esses dados. Eles não têm acesso a detalhes de implementação.
 - Comportamento semelhante acontece quando usamos as bibliotecas padrão do C++: `iostream`, `string`, `cstdlib`, `cmath`, etc.

Tipos Abstratos de Dados (TADs)

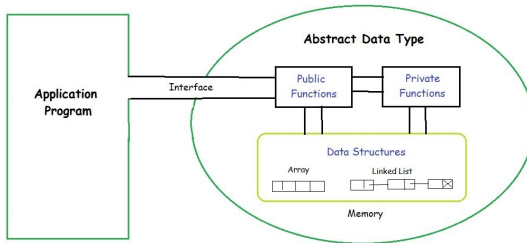


Imagem extraída de: www.geeksforgeeks.org

- A **interface** do TAD apenas menciona quais operações podem ser executadas, mas não como essas operações serão implementadas.

Tipos Abstratos de Dados (TADs)

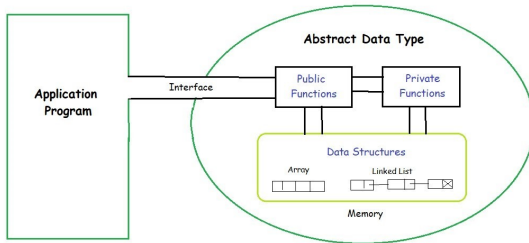


Imagem extraída de: www.geeksforgeeks.org

- A **interface** do TAD apenas menciona quais operações podem ser executadas, mas não como essas operações serão implementadas.
 - Não especifica como os dados serão organizados na memória e quais algoritmos serão usados para implementar as operações.

Tipos Abstratos de Dados (TADs)

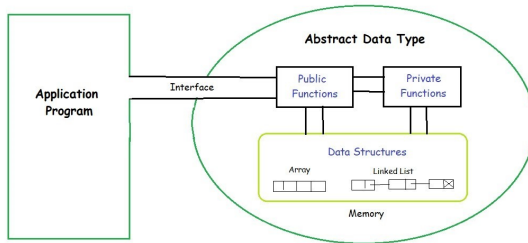


Imagem extraída de: www.geeksforgeeks.org

- A **interface** do TAD apenas menciona quais operações podem ser executadas, mas não como essas operações serão implementadas.
 - Não especifica como os dados serão organizados na memória e quais algoritmos serão usados para implementar as operações.
- É chamado de “abstrato” porque fornece uma visão independente da implementação. O processo de fornecer apenas o essencial e ocultar os detalhes é conhecido como **abstração**.

Características de um TAD

- Um TAD define o comportamento de um tipo de dado sem se preocupar com sua implementação. Entretanto, esta especificação não é reconhecida pelo computador.

Características de um TAD

- Um TAD define o comportamento de um tipo de dado sem se preocupar com sua implementação. Entretanto, esta especificação não é reconhecida pelo computador.
- É preciso criar uma **representação concreta** (através de um tipo concreto ou representacional) que nos diz:

Características de um TAD

- Um TAD define o comportamento de um tipo de dado sem se preocupar com sua implementação. Entretanto, esta especificação não é reconhecida pelo computador.
- É preciso criar uma **representação concreta** (através de um tipo concreto ou representacional) que nos diz:
 - como um TAD é implementado.

Características de um TAD

- Um TAD define o comportamento de um tipo de dado sem se preocupar com sua implementação. Entretanto, esta especificação não é reconhecida pelo computador.
- É preciso criar uma **representação concreta** (através de um tipo concreto ou representacional) que nos diz:
 - como um TAD é implementado.
 - como seus dados são colocados dentro do computador.

Características de um TAD

- Um TAD define o comportamento de um tipo de dado sem se preocupar com sua implementação. Entretanto, esta especificação não é reconhecida pelo computador.
- É preciso criar uma **representação concreta** (através de um tipo concreto ou representacional) que nos diz:
 - como um TAD é implementado.
 - como seus dados são colocados dentro do computador.
 - como estes dados são manipulados por suas operações (funções).

Características de TAD

- A chave para se conseguir verdadeiramente implementar tipos abstratos de dados é aplicar o conceito de **Independência de Representação**:

Características de TAD

- A chave para se conseguir verdadeiramente implementar tipos abstratos de dados é aplicar o conceito de **Independência de Representação**:
 - Um programa deveria ser projetado de forma que a representação de um tipo de dado possa ser modificada sem que isto interfira no restante do programa.

Características de TAD

- A chave para se conseguir verdadeiramente implementar tipos abstratos de dados é aplicar o conceito de **Independência de Representação**:
 - Um programa deveria ser projetado de forma que a representação de um tipo de dado possa ser modificada sem que isto interfira no restante do programa.
- A aplicação deste conceito é melhor realizada através da **modularização** do programa.

Módulos e compilação em separado

- **Cabeçalhos (*.h) e Unidades de tradução (*.cpp)**: contêm código-fonte
- **Preprocessador**: realiza substituição de texto
- **Compilador**: traduz UTs em arquivos objeto (.o)
- **‘‘Lincador’’**: linca arquivos objetos e bibliotecas externas em um arquivo executável

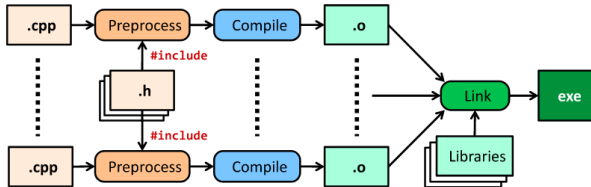


Imagem extraída de: https://hackingcpp.com/cpp/lang/separate_compilation.html

TAD Ponto



Exemplo de implementação de um TAD

- Vamos considerar a criação de um TAD para representar um ponto no espaço \mathbb{R}^2 .
- Para isso, devemos definir um tipo abstrato, que denominamos Ponto, e o conjunto de funções que operam sobre esse tipo.

Exemplo de implementação de um TAD

- Vamos considerar a criação de um TAD para representar um ponto no espaço \mathbb{R}^2 .
- Para isso, devemos definir um tipo abstrato, que denominamos Ponto, e o conjunto de funções que operam sobre esse tipo.
- Neste exemplo, vamos considerar as seguintes operações:
 - **cria**: cria um ponto com coordenada x e y
 - **libera**: libera a memória alocada por um ponto
 - **acessa**: devolve as coordenadas de um ponto
 - **atribui**: atribui novos valores às coordenadas de um ponto
 - **distancia**: calcula a distância entre dois pontos.

Implementação de um TAD

- Uma vez definido um TAD e especificadas as operações associadas, ele pode ser implementado em uma linguagem de programação.

Implementação de um TAD

- Uma vez definido um TAD e especificadas as operações associadas, ele pode ser implementado em uma linguagem de programação.
- Em **linguagens estruturadas**, como a C, a implementação é feita pela **definição de tipos** juntamente com a **implementação de funções**.

Implementação de um TAD

- Uma vez definido um TAD e especificadas as operações associadas, ele pode ser implementado em uma linguagem de programação.
- Em **linguagens estruturadas**, como a C, a implementação é feita pela **definição de tipos** juntamente com a **implementação de funções**.
- Em **linguagens orientadas a objeto** (C++, Java) a implementação de um TAD é naturalmente feita através de **classes**.

Implementação de um TAD

- Uma vez definido um TAD e especificadas as operações associadas, ele pode ser implementado em uma linguagem de programação.
- Em **linguagens estruturadas**, como a C, a implementação é feita pela **definição de tipos** juntamente com a **implementação de funções**.
- Em **linguagens orientadas a objeto** (C++, Java) a implementação de um TAD é naturalmente feita através de **classes**.
- Vamos implementar o TAD Ponto usando o paradigma de programação orientada a objetos.

Programação orientada a objetos



Objetos

- O mundo real é formado por objetos que interagem entre si (casa, carro, aluno, professor, etc.)

O que é um objeto? É qualquer coisa, real ou abstrata, com limites e significados bem definidos para a aplicação.

Possuem um **estado** (valores e atributos) e oferecem **operações** (comportamentos) para examinar ou alterar esse estado.



Imagens do site Pixabay



Objetos

- Então, um objeto possui estados (**atributos**) e operações (**funções**).
- Em C++, os atributos seriam as **variáveis** que guardam suas informações. E as funções (ou funções-membro), são funções usadas para interagir com esse objeto, como, por exemplo, uma função para mudar algum atributo.

Objetos

- Então, um objeto possui estados (**atributos**) e operações (**funções**).
- Em C++, os atributos seriam as **variáveis** que guardam suas informações. E as funções (ou funções-membro), são funções usadas para interagir com esse objeto, como, por exemplo, uma função para mudar algum atributo.

Porém: Objetos não são programados diretamente. Para criar um objeto, precisamos primeiramente definir uma CLASSE de objetos antes.

Objetos

- Então, um objeto possui estados (**atributos**) e operações (**funções**).
- Em C++, os atributos seriam as **variáveis** que guardam suas informações. E as funções (ou funções-membro), são funções usadas para interagir com esse objeto, como, por exemplo, uma função para mudar algum atributo.

Porém: Objetos não são programados diretamente. Para criar um objeto, precisamos primeiramente definir uma CLASSE de objetos antes.

- Por exemplo, todas as pessoas possuem atributos em comum como: altura, data de nascimento, cor dos olhos, tipo sanguíneo, etc. E podem realizar atividades comuns como: comer, respirar, dormir, etc.

Objetos

- Então, um objeto possui estados (**atributos**) e operações (**funções**).
- Em C++, os atributos seriam as **variáveis** que guardam suas informações. E as funções (ou funções-membro), são funções usadas para interagir com esse objeto, como, por exemplo, uma função para mudar algum atributo.

Porém: Objetos não são programados diretamente. Para criar um objeto, precisamos primeiramente definir uma CLASSE de objetos antes.

- Por exemplo, todos as pessoas possuem possuem atributos em comum como: altura, data de nascimento, cor dos olhos, tipo sanguíneo, etc. E podem realizar atividades comuns como: comer, respirar, dormir, etc.
- Logo, esses atributos e funções comuns são agrupados em uma classe Pessoa, responsável por modelar essa entidade.

Classes

- Uma **classe** em C++, é um tipo definido pelo usuário, assim como uma estrutura (struct).

Classes

- Uma **classe** em C++, é um tipo definido pelo usuário, assim como uma estrutura (struct).
- Uma classe é uma forma lógica de **encapsular dados** e **operações sobre dados** em uma mesma estrutura.

Classes

- Uma **classe** em C++, é um tipo definido pelo usuário, assim como uma estrutura (struct).
- Uma classe é uma forma lógica de **encapsular dados** e **operações sobre dados** em uma mesma estrutura.
- Assim que criamos uma classe, podemos INSTANCIAR um objeto, com seus respectivos atributos, que são individuais para cada objeto.



Definição de uma Classe em C++

```
1 class nome_da_classe {  
2     private:  
3         // Atributos  
4         int x, y;  
5     public:  
6         // Funcoes-membro  
7         int funcao ( int val ) {  
8             return (x * val + y);  
9         }  
10 };
```

- Por meio do encapsulamento, podemos decidir “como” a nossa classe interage com outras classes.

Definição de uma Classe em C++

```
1 class nome_da_classe {  
2     private:  
3         // Atributos  
4         int x, y;  
5     public:  
6         // Funcoes-membro  
7         int funcao ( int val ) {  
8             return (x * val + y);  
9         }  
10 };
```

- Por meio do encapsulamento, podemos decidir “como” a nossa classe interage com outras classes.
- Todas as classes em C++ possuem duas funções-membros chamadas **construtor** e **destrutor** que trabalham de maneira automática para assegurar que haja criação e remoção adequada de instâncias da classe, isto é, objetos.

Construtores

- Um **construtor** é uma função-membro que é executada automaticamente sempre que um objeto é criado.
- É geralmente utilizado para inicializar as variáveis dentro de um objeto, assim que ele é instanciado.

Implementando um construtor (1)

```
1 #include <iostream> // construtor.cpp
2
3 class Ponto {
4     private:
5         double x;
6         double y;
7     public:
8         Ponto(double X, double Y) {
9             x = X;
10            y = Y;
11        }
12
13        // construtor sem argumentos
14        Ponto() {
15            x = y = 0.0;
16        }
17 };
18
19 int main() {
20     // Instanciando um objeto chamando o construtor
21     Ponto p { 2.3, 4.5 };
22     Ponto p2;
23 }
```

Implementando um construtor (2)

```
1 #include <iostream> // construtor2.cpp
2
3 class Ponto {
4     private:
5         double x;
6         double y;
7     public:
8         // usando lista inicializadora de membros
9         Ponto(double X, double Y)
10             : x(X), y(Y) { }
11
12         // construtor sem argumentos
13         Ponto()
14             : Ponto(9, -100)
15         {
16             std::cout << x << "," << y << std::endl;
17         }
18 };
19
20 int main() {
21     // Instanciando um objeto chamando o construtor
22     Ponto p2;
23 }
```

Implementando um construtor (3)

```
1 #include <iostream> // construtor3.cpp
2
3 class Ponto {
4     private:
5         double x;
6         double y;
7     public:
8         // Construtor sem argumentos
9         // que chama outro construtor
10        Ponto()
11            : Ponto(-1,-1) { }
12
13        // usando lista inicializadora de membros
14        Ponto(double X, double Y)
15            : x(X), y(Y)
16            {
17                std::cout << "(" << x << "," << y << ")\n";
18            }
19 };
20
21 int main() {
22     Ponto p { 2.3, 4.5 };
23     Ponto p2;
24 }
```

Implementando um construtor (4)

```
1 #include <iostream> // construtor4.cpp
2
3 class Ponto {
4     private:
5         double x;
6         double y;
7         double z;
8     public:
9         // permite alguns argumentos nao serem fornecidos
10        Ponto(double X = 0, double Y = 0, double Z = 0)
11            : x(X), y(Y), z(Z)
12        {
13            std::cout << "(" << x << "," << y <<
14                "," << z << ")\n";
15        }
16 };
17
18 int main() {
19     Ponto p1 { 4, 5, 7 };
20     Ponto p2 { 4, 5 };
21     Ponto p3 { 4 };
22     Ponto p4;
23 }
```


Construtor default

Se você não criar um construtor, o compilador do C++ implementa um automaticamente (**construtor default**). Cada variável é então inicializada por default. Essa inicialização faz o seguinte:

- Atributos de tipo nativo (int, char, double, etc) possuem um valor indefinido após a inicialização por default. Elas ficam com o valor que existir na memória (lixo).
- Um objeto pertencente a uma certa classe é inicializado por default chamando o **construtor default**, que é aquele que não tem parâmetros. Se esse construtor não existir ou estiver inacessível (**private**), ocorre um erro de compilação.
- Um atributo do tipo array tem cada um de seus elementos inicializados como descrito nos itens acima.

Destrutores

- Sabe-se que o C++ já faz coleta automática das variáveis e dos objetos que não são alocados dinamicamente.
- Os destrutores servem para liberar os dados que foram alocados dinamicamente (usando o operador `new`)
- Lembre-se que, para liberar a memória alocada pela função `new`, usamos o operador `delete`.

Implementando um destrutor simples

```
1 #include <iostream> // destrutor.cpp
2
3 class Ponto {
4 private:
5     double x;
6     double y;
7
8 public:
9     Ponto(double X, double Y) {
10         x = X;
11         y = Y;
12         std::cout << "Ponto construido" << std::endl;
13     }
14
15     // Destrutor (note o til antes do nome da funcao)
16     ~Ponto() {
17         std::cout << "Ponto destruido" << std::endl;
18     }
19
20     double getX() { return x; }
21     double getY() { return y; }
22     void setX(double x) { this->x = x; }
23     void setY(double y) { this->y = y; }
24 };
```

Encapsulamento

- Muitas vezes não queremos que as outras classes tenham acesso direto aos atributos e funções específicas dos objetos de uma classe específica.
- A técnica responsável pelo controle de acesso aos elementos de uma classe é o **encapsulamento**
- Nós podemos controlar esse acesso usando os chamados “especificadores de acesso”.
- Os especificadores de acesso são conhecidos pelos identificadores: **public**, **protected** e **private**.

Especificadores de acesso

Esses especificadores modificam os direitos de acesso que as classes e funções externas têm sobre os elementos de uma classe.

Por enquanto, usaremos apenas o `public` e o `private`.

Especificadores de acesso

Esses especificadores modificam os direitos de acesso que as classes e funções externas têm sobre os elementos de uma classe.

Por enquanto, usaremos apenas o `public` e o `private`.

- Os membros `privados` (`private`) são acessíveis apenas pelos membros da própria classe.

Especificadores de acesso

Esses especificadores modificam os direitos de acesso que as classes e funções externas têm sobre os elementos de uma classe.

Por enquanto, usaremos apenas o **public** e o **private**.

- Os membros **privados** (`private`) são acessíveis apenas pelos membros da própria classe.
- Os membros **públicos** (`public`) são acessíveis através de qualquer classe ou função que interage com os objetos dessa classe.

getters e setters

- Para que possamos acessar os valores de atributos privados de uma classe, devemos criar funções-membro específicas para fazer isso, chamadas **getters** e **setters**.

getters e setters

- Para que possamos acessar os valores de atributos privados de uma classe, devemos criar funções-membro específicas para fazer isso, chamadas **getters** e **setters**.
- **Setters**: Modificam os dados do objeto.
- **Getters**: Acessam os valores, mas não permitem modificá-los.

Implementação do TAD Ponto como classe



Lembrando interface

- Vamos considerar a criação de um TAD para representar um ponto no espaço \mathbb{R}^2 .
- Para isso, devemos definir um tipo abstrato, que denominamos Ponto, e o conjunto de funções que operam sobre esse tipo.

Lembrando interface

- Vamos considerar a criação de um TAD para representar um ponto no espaço \mathbb{R}^2 .
- Para isso, devemos definir um tipo abstrato, que denominamos Ponto, e o conjunto de funções que operam sobre esse tipo.
- Neste exemplo, vamos considerar as seguintes operações:
 - **cria**: cria um ponto com coordenada x e y
 - **libera**: libera a memória alocada por um ponto
 - **acessa**: devolve as coordenadas de um ponto
 - **atribui**: atribui novos valores às coordenadas de um ponto
 - **distancia**: calcula a distância entre dois pontos.

Arquivo Ponto2.h

```
1 #ifndef PONTO_H
2 #define PONTO_H
3 #include <iostream>
4 #include <cmath>
5 using namespace std;
6
7 class Ponto {
8 private:
9     double x;
10    double y;
11 public:
12    // Construtor
13    Ponto() {
14        this->x = 0;
15        this->y = 0;
16    }
17
18    Ponto(double X, double Y = 0) {
19        this->x = X;
20        this->y = Y;
21    }
```

Final do arquivo Ponto2.h

```
1 // Destrutor
2 ~Ponto() {
3     cout << "Ponto destruido" << x << "," << y << endl;
4 }
5
6 // Getters
7 double getX() { return x; }
8 double getY() { return y; }
9
10 // Setters
11 void setX(double X) { x = X; }
12 void setY(double Y) { y = Y; }
13
14 // Calcula a distancia entre dois pontos:
15 // Entre o ponto que chamou essa funcao
16 // e o ponto p passado como parametro
17 double distancia(Ponto p) {
18     double dx = this->x - p.x;
19     double dy = this->y - p.x;
20     return sqrt(dx*dx + dy*dy);
21 }
22 };
23
24 #endif
```

Programa Cliente — main2.cpp

```
1 #include <iostream> // main2.cpp
2 #include "Ponto2.h"
3 using namespace std;
4
5 int main() {
6     Ponto p1 { 2.3, 4.5 };
7     Ponto p2 { 4, 7.8 };
8     Ponto p3 = p2;
9
10    cout << "Ponto 1: ";
11    cout << "(" << p1.getX() << "," << p1.getY() << ")\n";
12
13    cout << "Ponto 2: ";
14    cout << "(" << p2.getX() << "," << p2.getY() << ")\n";
15
16    cout << "Ponto 3: ";
17    cout << "(" << p3.getX() << "," << p3.getY() << ")\n";
18
19    cout << "Distancia: " << p1.distancia(p2) << endl;
20    return 0;
21 }
```

Outra Implementação do TAD Ponto



Arquivo Ponto3.h

```
1  #ifndef PONT03_H
2  #define PONT03_H
3
4  class Ponto {
5      private:
6          double x;
7          double y;
8      public:
9          Ponto();
10         Ponto(double X, double Y);
11
12         ~Ponto();
13
14         double getX();
15         double getY();
16
17         void setX(double X);
18         void setY(double Y);
19
20         double distancia(Ponto p);
21 };
22
23 #endif
```

Arquivo Ponto3.cpp

```
1 #include <iostream>
2 #include <cmath>
3 #include "Ponto3.h"
4
5 Ponto::Ponto() {
6     this->x = 0;
7     this->y = 0;
8 }
9
10 Ponto::Ponto(double X, double Y) {
11     this->x = X;
12     this->y = Y;
13 }
14
15 Ponto::~Ponto() {
16     std::cout << "Ponto destruido" << std::endl;
```

Final do Arquivo Ponto3.cpp

```
17
18 double Ponto::getX() { return x; }
19 double Ponto::getY() { return y; }
20
21 void Ponto::setX(double X) { x = X; }
22 void Ponto::setY(double Y) { y = Y; }
23
24 double Ponto::distancia(Ponto p) {
25     double dx = x - p.x;
26     double dy = y - p.y;
27     return sqrt(dx*dx + dy*dy);
28 }
```

Exercícios



Exercício — TAD Círculo

- Vamos considerar a criação de um tipo de dado para representar um círculo no \mathbb{R}^2 .
- Implemente o TAD por meio de uma classe chamada `Circulo`. Sua classe deve ter os seguintes métodos:
 - o construtor `Circulo(float raio, Ponto centro)`: cria um círculo cujo centro é um atributo do tipo `Ponto` e raio é um `float`;
 - void `setRaio(float r)`: atribui novo valor ao raio do círculo;
 - float `getRaio()` obtém o raio.
 - `Ponto getCentro()`: obtém o centro.
 - float `area()`: calcula a área do círculo.
 - bool `interior(Ponto p)`: verifica se o `Ponto p` está dentro do círculo.

Exercício — TAD Fração

- O TAD Fração pode ser implementado como uma Classe chamada `Fracao`. A classe deve ter os atributos `numerador` e `denominador`, e deve ter os seguintes métodos:
 - o construtor `Fracao(N, D)`: recebe dois inteiros N e D como argumento e retorna a fração $\frac{N}{D}$.
 - `float numerador()`: retorna o numerador.
 - `float denominador()`: retorna o denominador.
 - `float soma(F2)`: recebe a fração $F2$ como argumento e retorna a fração resultante da soma da fração em questão com a fração $F2$.

Exercício — TAD Matriz

- Implementar em C++ um TAD chamado Matriz.
- O TAD Matriz encapsula uma matriz com n linhas e m colunas sobre a qual podemos fazer as seguintes operações:
 - criar matriz alocada dinamicamente
 - destruir a matriz alocada dinamicamente
 - acessar valor na posição (i, j) da matriz
 - atribuir valor ao elemento na posição (i, j)
 - retornar o número de linhas da matriz
 - retornar o número de colunas da matriz
 - imprimir a matriz na tela do terminal

FIM

