TAD - Tipos Abstratos de Dados

Estrutura de Dados — QXD0010

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

Universidade Federal do Ceará
Campus Quixadá

2º semestre/2023

Introdução

Tipos Abstratos de Dados (TADs)

- Um Tipo Abstrato de Dados (TAD) é uma especificação de um **conjunto de dados** e **operações** que podem ser executadas sobre esses dados.
  - **TAD** = dados + operações

Tipos Abstratos de Dados (TADs)

- Um Tipo Abstrato de Dados (TAD) é uma especificação de um **conjunto de dados** e **operações** que podem ser executadas sobre esses dados.
  - **TAD** = dados + operações
- 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 Dados (TAD) é uma especificação de um **conjunto de dados** e **operações** que podem ser executadas sobre esses dados.
  - **TAD** = dados + operações
- 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.

Características de um TAD

Um tipo abstrato de dados possui duas partes principais:

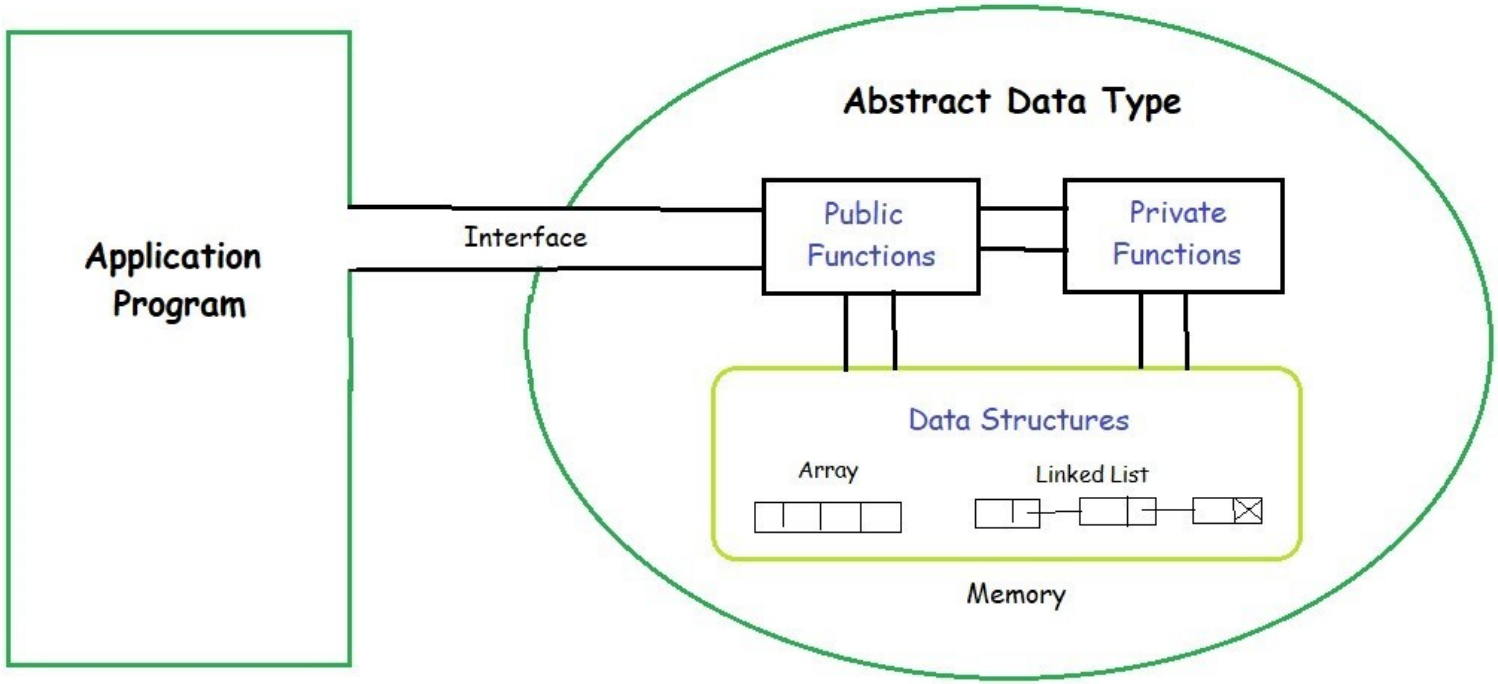
- **Interface:** define o comportamento do TAD, como as operações do tipo podem ser executadas, mas não como essas operações são implementadas.
  - Não especifica como os dados serão organizados na memória e quais algoritmos serão usados para implementar as operações.

Características de um TAD

Um tipo abstrato de dados possui duas partes principais:

- **Interface:** define o comportamento do TAD, como as operações do tipo podem ser executadas, mas não como essas operações são implementadas.
  - Não especifica como os dados serão organizados na memória e quais algoritmos serão usados para implementar as operações.
- **Representação concreta:** é a implementação em si, que nos diz:
  - como um TAD foi implementado.
  - como seus dados são colocados dentro do computador.
  - como estes dados são manipulados por suas operações (funções).

Tipos Abstratos de Dados (TADs)



- 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.

Como implementar um TAD?

- A chave para se conseguir 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.

Como implementar um TAD?

- A chave para se conseguir 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 (em programação estruturada)
  - criação de classes (em programação orientada a objetos)

Como implementar um TAD?

- A chave para se conseguir 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 (em programação estruturada)
  - criação de classes (em programação orientada a objetos)

- Neste curso, vamos usar **classes** para implementar TADs. Existem dezenas de bons livros que usam C e programação estruturada para implementar TADs, procure um deles para ver outra abordagem!

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Objetos e Classes em C++

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 atributos (dados), um estado (valores dos atributos) e oferecem operações (comportamentos) para examinar ou alterar esse estado.

Objetos

- Então, um objeto possui dados (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 dados (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 dados (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 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 dados (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 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).

- Uma classe é uma forma lógica de encapsular dados e operações em uma mesma estrutura lógica.

- Um objeto é uma **instância** de uma classe. 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++

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  class nome_da_classe {
2  private:
3      // Atributos
4      int x;
5      int y;
6
7  public:
8      // Funcoes-membro
9      int funcao( int val ) {
10         return x * val + y;
11     }
12 };
```

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

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 modificadores de acesso.

Modificadores de acesso

Modificadores de acesso: Alteram os direitos de acesso que as classes e funções externas têm sobre os membros de uma classe.

Os modificadores de acesso mais usados são: `public` e `private`.

Modificadores de acesso

Modificadores de acesso: Alteram os direitos de acesso que as classes e funções externas têm sobre os membros de uma classe.

Os modificadores de acesso mais usados são: `public` e `private`.

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

Modificadores de acesso

Modificadores de acesso: Alteram os direitos de acesso que as classes e funções externas têm sobre os membros de uma classe.

Os modificadores de acesso mais usados são: `public` e `private`.

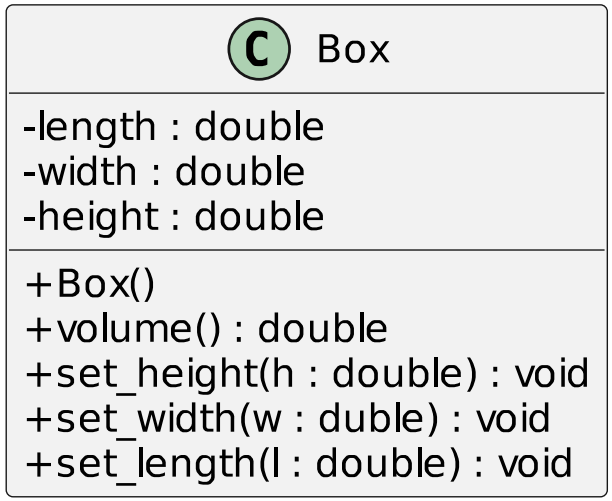
- Os membros **privados** (private) são acessíveis apenas pelos membros da própria classe.

- Os membros **públicos** (public) são acessíveis dentro da classe e através de qualquer classe ou função que interage com os objetos dessa classe.

Exemplo: Implementando uma caixa

- Uma Caixa é um objeto que tem comprimento, largura e altura.
- Todo objeto do tipo Caixa deve fornecer um modo seguro de podermos ajustar suas três medidas, sempre que quisermos.
- Todo objeto do tipo Caixa sabe calcular seu volume e possui uma função-membro que retorna esse valor.
- Vamos implementar uma caixa como uma classe, a fim de que possamos instanciar objetos do tipo Caixa para podermos usar em um programa.

Diagrama UML inicial da classe Caixa



Exemplo: Implementando uma caixa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Arquivo Box1.h
2  #ifndef BOX1_H
3  #define BOX1_H
4
5  class Box {
6  private:
7      double length {1.0}; // comprimento
8      double width {1.0};  // largura
9      double height {1.0}; // altura
10
11 public:
12     double volume() {
13         return length * width * height;
14     }
15 };
16
17 #endif
```

Exemplo: Implementando uma caixa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Arquivo mainBox1.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box1.h"
5  using namespace std;
6
7  int main() {
8      Box box; // intancia o objeto
9
10     cout << fixed << setprecision(2)
11          << box.volume() << endl; // imprime o volume: 1.00
12 }
```

Exemplo: Implementando uma caixa

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Arquivo mainBox1.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box1.h"
5  using namespace std;
6
7  int main() {
8      Box box; // intancia o objeto
9
10     cout << fixed << setprecision(2)
11          << box.volume() << endl; // imprime o volume: 1.00
12 }
```

- E se quiséssemos instanciar a caixa com dimensões estabelecidas no momento da instanciação e não com os valores pré-estabelecidos de fábrica?

Construtores

- Um construtor é uma função-membro que é invocada automaticamente sempre que um objeto é criado. Um construtor não tem tipo de retorno.

- É geralmente utilizado para inicializar as variáveis dentro de um objeto, assim que ele é instanciado.

Construtores

- Um construtor é uma função-membro que é invocada automaticamente sempre que um objeto é criado. Um construtor não tem tipo de retorno.

- É geralmente utilizado para inicializar as variáveis dentro de um objeto, assim que ele é instanciado.

- **Toda** classe em C++ possui pelo menos um construtor. Se você não definir um construtor para a sua classe, o C++ define um automaticamente por você, é o chamado construtor default. Esse construtor não recebe parâmetro.

- Se você quiser um contrutor que receba parâmetros, você mesmo deve escrever um. **Atenção:** a partir do momento que você cria tal construtor, o C++ deixa de criar o construtor default!

Implementando um construtor (1)


```
1  // Arquivo Box2.h
2  #ifndef BOX2_H
3  #define BOX2_H
4
5  class Box {
6  private:
7      double length {1.0};   // comprimento
8      double width {1.0};    // largura
9      double height {1.0};   // altura
10
11 public:
12     Box(double lv, double wv, double hv) { // construtor
13         length = lv;
14         width = wv;
15         height = hv;
16     }
17
18     double volume() { // calcula volume da caixa
19         return length * width * height;
20     }
21 };
22
23 #endif
```

Implementando um construtor (1)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1
2  // Arquivo mainBox2.cpp
3  #include <iostream>
4  #include <iomanip>
5  #include "Box2.h"
6  using namespace std;
7
8  int main() {
9      Box abox {2.0, 3.0, 4.0}; // intancia o objeto
10     Box bbox; // ERRO: nao existe construtor default
11
12     cout << fixed << setprecision(2)
13          << abox.volume() << endl; // imprime o volume: 24.00
14 }
```

Implementando um construtor (2)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Usando a palavra-chave default

```
1  class Box { // Arquivo Box3.h
2  private:
3      double length {1.0};   // comprimento
4      double width {1.0};    // largura
5      double height {1.0};   // altura
6
7  public:
8      // instrui compilador a criar um construtor default
9      Box() = default;
10
11     // construtor com tres parametros
12     Box(double lv, double wv, double hv) {
13         length = lv;
14         width = wv;
15         height = hv;
16     }
17
18     double volume() { // calcula volume da caixa
19         return length * width * height;
20     }
21 };
```

Implementando um construtor (2)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Arquivo mainBox3.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box3.h"
5  using namespace std;
6
7  int main() {
8      Box abox {2.0, 3.0, 4.0}; // intancia o objeto
9      Box bbox;
10
11     cout << fixed << setprecision(2);
12     cout << abox.volume() << endl; // imprime o volume: 24.00
13     cout << bbox.volume() << endl; // imprime o volume:  1.00
14 }
```

Implementando um construtor (3)

Parâmetros com valores default

```
1  class Box { // Arquivo Box4.h
2  private:
3      double length;
4      double width;
5      double height;
6
7  public:
8      // permite que os argumentos nao sejam fornecidos no
9      // momento da instanciacao
10     Box(double lv = 1.0, double wv = 1.0, double hv = 1.0) {
11         length = lv;
12         width = wv;
13         height = hv;
14     }
15     // calcula volume da caixa
16     double volume() {
17         return length * width * height;
18     }
19 };
```

Implementando um construtor (3)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // arquivo mainBox4.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box4.h"
5  using namespace std;
6
7  int main() {
8      Box abox;
9      Box bbox {2.0};
10     Box cbox {2.0, 3.0};
11     Box dbox {2.0, 3.0, 4.0};
12
13     cout << fixed << setprecision(2);
14     cout << abox.volume() << endl; // 1.0
15     cout << bbox.volume() << endl; // 2.0
16     cout << cbox.volume() << endl; // 6.0
17     cout << dbox.volume() << endl; // 24.0
18 }
```

Implementando um construtor (4)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Usando uma lista inicializadora

```
1  class Box { // Arquivo Box5.h
2  private:
3      double length {1.0};   // comprimento
4      double width {1.0};    // largura
5      double height {1.0};   // altura
6
7  public:
8      Box() = default;  // construtor default
9
10     // construtor usando uma lista inicializadora
11     Box(double lv, double wv, double hv)
12         : length{lv}, width{wv}, height{hv}
13     {
14     }
15
16     // calcula volume da caixa
17     double volume() {
18         return length * width * height;
19     }
20 };
```

Implementando um construtor (4)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // arquivo mainBox5.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box5.h"
5  using namespace std;
6
7  int main() {
8      Box abox;
9      Box bbox {2.0, 3.0, 4.0};
10
11     cout << fixed << setprecision(2);
12     cout << abox.volume() << endl; // 1.0
13     cout << bbox.volume() << endl; // 24.0
14 }
```

Vantagem de usar lista inicializadora no construtor

- Quando você inicializa um atributo no corpo do construtor usando uma atribuição (=), primeiro o atributo é criado (se for uma instância de classe, um construtor default é invocado), somente depois disso é que a atribuição é executada, como uma operação separada.

- Quando você usa uma lista inicializadora, o valor é usado para inicializar o atributo ao mesmo tempo em que ele é criado. Esse processo pode ser mais eficiente, principalmente se o atributo for uma instância de uma classe.

Implementando um construtor (5)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Delegando construtores

```
1  class Box { // Arquivo Box6.h
2  private:
3      double length;
4      double width;
5      double height;
6
7  public:
8      Box() // construtor que invoca outro construtor
9          : Box{1.0, 1.0, 1.0}
10     {}
11
12     // construtor usando uma lista inicializadora
13     Box(double lv, double wv, double hv)
14         : length{lv}, width{wv}, height{hv}
15     {}
16
17     // calcula volume da caixa
18     double volume() {
19         return length * width * height;
20     }
21 };
```

Implementando um construtor (5)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // arquivo mainBox6.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box6.h"
5  using namespace std;
6
7  int main() {
8      Box abox;
9      Box bbox {2.0, 3.0, 4.0};
10
11     cout << fixed << setprecision(2);
12     cout << abox.volume() << endl; // 1.0
13     cout << bbox.volume() << endl; // 24.0
14 }
```

Implementando funções-membro fora da classe


```
1  #ifndef BOX8_H  // Arquivo Box8.h
2  #define BOX8_H
3
4  class Box {
5  private:
6      double length;
7      double width;
8      double height;
9
10 public:
11     Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);
12     double volume();
13 };
14
15 Box::Box(double lv, double wv, double hv)
16     : length{lv}, width{wv}, height{hv}
17 {}
18
19 double Box::volume() {
20     return length * width * height;
21 }
22
23 #endif
```

Observações

- Em C++, ao implementar as funções-membro da classe, é comum colocar a declaração na classe num arquivo de cabeçalho (arquivo com extensão .h) e colocar a implementação das funções-membro em um arquivo-fonte (arquivo com extensão .cpp)

Observações

- Em C++, ao implementar as funções-membro da classe, é comum colocar a declaração na classe num arquivo de cabeçalho (arquivo com extensão .h) e colocar a implementação das funções-membro em um arquivo-fonte (arquivo com extensão .cpp)
  - Exemplo: Box.h e Box.cpp

Observações

- Em C++, ao implementar as funções-membro da classe, é comum colocar a declaração na classe num arquivo de cabeçalho (arquivo com extensão .h) e colocar a implementação das funções-membro em um arquivo-fonte (arquivo com extensão .cpp)
  - Exemplo: Box.h e Box.cpp

- Os programas que quiserem utilizar a classe só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Box.h

Observações

- Em C++, ao implementar as funções-membro da classe, é comum colocar a declaração na classe num arquivo de cabeçalho (arquivo com extensão .h) e colocar a implementação das funções-membro em um arquivo-fonte (arquivo com extensão .cpp)
  - Exemplo: Box.h e Box.cpp

- Os programas que quiserem utilizar a classe só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Box.h

- O arquivo de implementação da classe (o arquivo Box.cpp) deve sempre incluir o arquivo de interface da classe.

Observações

- Em C++, ao implementar as funções-membro da classe, é comum colocar a declaração na classe num arquivo de cabeçalho (arquivo com extensão .h) e colocar a implementação das funções-membro em um arquivo-fonte (arquivo com extensão .cpp)
  - Exemplo: Box.h e Box.cpp

- Os programas que quiserem utilizar a classe só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Box.h

- O arquivo de implementação da classe (o arquivo Box.cpp) deve sempre incluir o arquivo de interface da classe.

- Isto é necessário por duas razões:

Observações

- Em C++, ao implementar as funções-membro da classe, é comum colocar a declaração na classe num arquivo de cabeçalho (arquivo com extensão .h) e colocar a implementação das funções-membro em um arquivo-fonte (arquivo com extensão .cpp)
  - Exemplo: Box.h e Box.cpp

- Os programas que quiserem utilizar a classe só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Box.h

- O arquivo de implementação da classe (o arquivo Box.cpp) deve sempre incluir o arquivo de interface da classe.

- Isto é necessário por duas razões:
  - Existem definições na interface que são necessárias na implementação.

Observações

- Em C++, ao implementar as funções-membro da classe, é comum colocar a declaração na classe num arquivo de cabeçalho (arquivo com extensão .h) e colocar a implementação das funções-membro em um arquivo-fonte (arquivo com extensão .cpp)
  - Exemplo: Box.h e Box.cpp

- Os programas que quiserem utilizar a classe só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Box.h

- O arquivo de implementação da classe (o arquivo Box.cpp) deve sempre incluir o arquivo de interface da classe.

- Isto é necessário por duas razões:
  - Existem definições na interface que são necessárias na implementação.
  - Precisamos garantir que as funções implementadas correspondem às funções da interface. O compilador verifica se os parâmetros das funções implementadas equivalem aos parâmetros dos protótipos.

Box9.h


```
1  // Arquivo Box9.h
2  #ifndef BOX9_H
3  #define BOX9_H
4
5  class Box {
6  private:
7      double length;
8      double width;
9      double height;
10
11 public:
12     Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);
13     double volume();
14 };
15
16 #endif
```

Box9.cpp


```
1 #include "Box9.h"
2
3 Box::Box(double lv, double wv, double hv)
4     : length{lv}, width{wv}, height{hv}
5 {}
6
7 double Box::volume() {
8     return length * width * height;
9 }
```

mainBox9.cpp


```
1  // arquivo mainBox9.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box9.h"
5  using namespace std;
6
7  int main() {
8      Box abox {2.0, 3.0, 4.0};
9      Box bbox { abox };
10
11     cout << fixed << setprecision(2);
12     cout << abox.volume() << endl; // 24.00
13     cout << bbox.volume() << endl; // 24.00
14 }
```

mainBox9.cpp


```
1  // arquivo mainBox9.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box9.h"
5  using namespace std;
6
7  int main() {
8      Box abox {2.0, 3.0, 4.0};
9      Box bbox { abox };
10
11     cout << fixed << setprecision(2);
12     cout << abox.volume() << endl; // 24.00
13     cout << bbox.volume() << endl; // 24.00
14 }
```

Como compilar no terminal do Linux:

```
g++ Box9.cpp mainBox9.cpp -o main
```

ou

```
g++ *.cpp -o main
```

Compilação separada

Temos três arquivos: Box9.h, Box9.cpp e mainBox9.cpp

1. Gerando o arquivo objeto Box9.o
   ```
   g++ -c Box9.cpp
   ```
2. Gerando o arquivo objeto mainBox9.o
   ```
   g++ -c mainBox9.cpp
   ```
3. Fazendo a linkagem e gerando o executável
   ```
   g++ Box9.o mainBox9.o -o main
   ```

Compilação separada

Temos três arquivos: Box9.h, Box9.cpp e mainBox9.cpp

1. Gerando o arquivo objeto Box9.o
   `g++ -c Box9.cpp`
2. Gerando o arquivo objeto mainBox9.o
   `g++ -c mainBox9.cpp`
3. Fazendo a linkagem e gerando o executável
   `g++ Box9.o mainBox9.o -o main`

Essas operações podem ser automatizadas usando um arquivo `Makefile`:

```
1 cliente: Box9.o mainBox9.o
2   g++ Box9.o mainBox9.o -o main && ./main
3 mainBox9.o: mainBox9.cpp
4   g++ -c mainBox9.cpp
5 Box9.o: Box9.cpp Box9.h
6   g++ -c Box9.cpp
```

Destrutor

- Destrutor é uma função-membro especial que é sempre invocada quando o objeto é liberado.
- Se você não implementar um destrutor para a classe, mesmo assim o C++ implementará um destrutor por você.
- Uma utilidade do destrutor é liberar memória que foi alocada dinamicamente dentro do objeto (usando o operador new)
- Assim como o construtor, o destrutor possui o mesmo nome que a classe, porém é antecedido pelo símbolo $\sim$ (til)

Implementando um destrutor simples

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream>
2
3  class Box { // Arquivo Box10.h
4  private:
5      double length;
6      double width;
7      double height;
8
9  public:
10     Box(double lv = 1.0, double wv = 1.0, double hv = 1.0)
11         : length{lv}, width{wv}, height{hv}
12     {}
13
14     // Destrutor
15     ~Box() { std::cout << "Box liberada.\n"; }
16
17     double volume() {
18         return length * width * height;
19     }
20 };
```

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.

Implementando getters e setters


```
1  // Arquivo Box11.h
2  #ifndef BOX11_H
3  #define BOX11_H
4
5  class Box {
6  private:
7      double length, width, height;
8
9  public:
10     Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);
11     double volume();
12
13     double getLength() { return length; }
14     double getWidth() { return width; }
15     double getHeight() { return height; }
16
17     void setLength(double lv) { if(lv > 0) length = lv; }
18     void setWidth(double wv) { if(wv > 0) width = wv; }
19     void setHeight(double hv) { if(hv > 0) height = hv; }
20 };
21
22 #endif
```

Implementando getters e setters (cont.)


```
1  // arquivo Box11.cpp
2  #include "Box11.h"
3
4  Box::Box(double lv, double wv, double hv)
5      : length{lv}, width{wv}, height{hv}
6  {}
7
8  double Box::volume() {
9      return length * width * height;
10 }
```

Implementando getters e setters (cont.)


```
1  // arquivo mainBox11.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box11.h"
5  using namespace std;
6
7  int main() {
8      Box box;
9
10     box.setLength(2.0);
11     box.setWidth(3.0);
12     box.setHeight(4.0);
13
14     cout << fixed << setprecision(2);
15     cout << "Length: " << box.getLength() << endl;
16     cout << "Width: " << box.getWidth() << endl;
17     cout << "Height: " << box.getHeight() << endl;
18     cout << "Volume: " << box.volume() << endl;
19 }
```

Ponteiro `this`

Como os objetos 'enxergam' as funções e atributos de uma classe?

1. Cada objeto tem seus próprios atributos.
2. Todos os objetos de uma dada classe compartilham uma única cópia das funções-membro.

Ponteiro `this`

Como os objetos 'enxergam' as funções e atributos de uma classe?

1. Cada objeto tem seus próprios atributos.
2. Todos os objetos de uma dada classe compartilham uma única cópia das funções-membro.

Questão: Se existe apenas uma cópia de cada função-membro e cada uma é usada por vários objetos, como os atributos apropriados são acessados e atualizados?

Ponteiro `this`

Como os objetos 'enxergam' as funções e atributos de uma classe?

1. Cada objeto tem seus próprios atributos.
2. Todos os objetos de uma dada classe compartilham uma única cópia das funções-membro.

Questão: Se existe apenas uma cópia de cada função-membro e cada uma é usada por vários objetos, como os atributos apropriados são acessados e atualizados?

- O compilador fornece um ponteiro implícito chamado `this` junto com os nomes das funções.
- O ponteiro `this` é passado como um argumento oculto para todas as chamadas de funções não-estáticas e está disponível como uma variável local dentro do corpo de todas as funções não-estáticas.

Ponteiro this — Exemplo 1


```
1  #ifndef BOX12_H // Arquivo Box12.h
2  #define BOX12_H
3
4  class Box {
5  private:
6      double length, width, height;
7
8  public:
9      Box(double lv = 1.0, double wv = 1.0, double hv = 1.0)
10         : length{lv}, width{wv}, height{hv} { }
11
12     double volume() { return length * width * height; }
13
14     double getLength() { return length; }
15     double getWidth() { return width; }
16     double getHeight() { return height; }
17
18     void setLength(double length) { this->length = length; }
19     void setWidth(double width) { this->width = width; }
20     void setHeight(double height) { this->height = height; }
21 };
22
23 #endif
```

Ponteiro `this` — Exemplo 1 (cont.)


```
1  // arquivo mainBox12.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box12.h"
5  using namespace std;
6
7  int main() {
8      Box box;
9
10     box.setLength(2.0);
11     box.setWidth(3.0);
12     box.setHeight(4.0);
13
14     cout << fixed << setprecision(2);
15     cout << "Length: " << box.getLength() << endl;
16     cout << "Width: " << box.getWidth() << endl;
17     cout << "Height: " << box.getHeight() << endl;
18     cout << "Volume: " << box.volume() << endl;
19 }
```

Ponteiro `this` — Exemplo 2

Retornando um ponteiro nos setters

```
1  #ifndef BOX13_H // Arquivo Box13.h
2  #define BOX13_H
3
4  class Box {
5  private:
6      double length, width, height;
7
8  public:
9      Box(double lv = 1.0, double wv = 1.0, double hv = 1.0)
10         : length{lv}, width{wv}, height{hv}
11     {}
12
13     double volume() { return length * width * height; }
14
15     double getLength() { return length; }
16     double getWidth() { return width; }
17     double getHeight() { return height; }
```

Ponteiro `this` — Exemplo 2 (cont.)

Retornando um ponteiro nos setters

```
18      Box* setLength(double length) {
19          this->length = length;
20          return this;
21      }
22
23      Box* setWidth(double width) {
24          this->width = width;
25          return this;
26      }
27
28      Box* setHeight(double height) {
29          this->height = height;
30          return this;
31      }
32  };
33
34  #endif
```

Ponteiro `this` — Exemplo 2 (cont.)

Retornando um ponteiro nos setters

```
1  // arquivo mainBox13.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box13.h"
5  using namespace std;
6
7  int main() {
8      Box box;
9
10     // Atencao: agora eh possivel encadear chamadas
11     box.setLength(2.0)->setWidth(3.0)->setHeight(4.0);
12
13     cout << fixed << setprecision(2);
14     cout << "Length: " << box.getLength() << endl;
15     cout << "Width: " << box.getWidth() << endl;
16     cout << "Height: " << box.getHeight() << endl;
17     cout << "Volume: " << box.volume() << endl;
18 }
```

Exercícios

TAD Point

- Criar de um TAD para representar um ponto (x, y) no espaço $\mathbb{R}^2$.

- Todo ponto no plano $\mathbb{R}^2$ possui é determinado por duas coordenadas. Estes são os dois atributos de um ponto.

- Consideramos as seguintes operações:
  - criar um ponto com coordenadas x e y
  - se for necessário, liberar a memória alocada por um ponto
  - devolver a coordenada x de um ponto
  - devolver a coordenada y de um ponto
  - atribuir novo valor à coordenada x do ponto
  - atribuir novo valor à coordenada y do ponto
  - calcular a distância entre dois pontos.

Exercício: Implemente um TAD chamado `Point` como uma classe, seguindo os requisitos listados acima.

Exercício — TAD Matrix

- Criar um TAD para representar uma matriz com n linhas e m colunas.
  Os valores n e m são determinados no momento da criação da matriz.

- Implemente o TAD por meio de uma classe chamada `Matrix`.
  Esse TAD 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
  - comparar a matriz com outra e decidir se são ou não iguais.

Exercício — TAD Circle

- Criar um TAD para representar um círculo no $\mathbb{R}^2$.
- Implemente o TAD por meio de uma classe chamada `Circle`. Todo círculo pode ser definido a partir do seu **centro** e do seu **raio**.
- Sua classe deve ter as seguintes funções-membro:
  - o construtor `Circle(double radius, Ponto& center)`: cria um círculo cujo centro é um atributo do tipo `Ponto` e raio é um `double`.
  - `void setRadius(double r)`: atribui novo valor ao raio do círculo.
  - `void setCenterX(double x)`: atribui novo valor ao x do centro.
  - `void setCenterY(double y)`: atribui novo valor ao y do centro.
  - `void setCenter(const Ponto& p)`: muda o centro.
  - `double getRadius()` obtém o raio.
  - `Ponto getCenter()`: obtém o centro.
  - `double area():` retorna a área do círculo.
  - `bool contains(const Point& p):` verifica se o ponto p está dentro do círculo.

Construtor de Cópia

Construtor de cópia (*copy construtor*)

- Além do construtor default, **toda** classe em C++ tem um construtor chamado construtor de cópia (*copy constructor*).

Construtor de cópia (*copy construtor*)

- Além do construtor default, **toda** classe em C++ tem um construtor chamado construtor de cópia (*copy constructor*).

- Esse construtor permite que eu possa inicializar um objeto passando como parâmetro para o construtor um objeto da mesma classe que já tenha sido criado previamente. Exemplo:

```
Box b1 { 1.0, 2.0, 3.0 };
Box b2 { b1 }; \\ chama o copy constructor
```

Construtor de cópia (*copy construtor*)

- Além do construtor default, **toda** classe em C++ tem um construtor chamado construtor de cópia (*copy constructor*).

- Esse construtor permite que eu possa inicializar um objeto passando como parâmetro para o construtor um objeto da mesma classe que já tenha sido criado previamente. Exemplo:

```
Box b1 { 1.0, 2.0, 3.0 };
Box b2 { b1 }; \\ chama o copy constructor
```

- O construtor de cópia recebe como parâmetro uma `const reference` para um objeto da mesma classe. Esse construtor realiza uma cópia campo-a-campo dos atributos do objeto passado como argumento.
  - Isso pode ser potencialmente perigoso se um atributo da classe for um ponteiro!

Implementando um copy constructor

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  class Box { // Arquivo Box7.h
2  private:
3      double length;
4      double width;
5      double height;
6
7  public:
8      Box(double lv = 1.0, double wv = 1.0, double hv = 1.0)
9          : length{lv}, width{wv}, height{hv}
10     {}
11
12     Box(const Box& b) // copy constructor
13         : length{b.length}, width{b.width}, height{b.height}
14     {}
15
16     // calcula volume da caixa
17     double volume() {
18         return length * width * height;
19     }
20 };
```

Implementando um copy constructor

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // arquivo mainBox7.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box7.h"
5  using namespace std;
6
7  int main() {
8      Box abox {2.0, 3.0, 4.0};
9      Box bbox { abox };
10
11     cout << fixed << setprecision(2);
12     cout << abox.volume() << endl; // 24.00
13     cout << bbox.volume() << endl; // 24.00
14 }
```

Ponteiro `this` — Exemplo 3

Retornando uma referência nos setters

```
1  #ifndef BOX14_H // Arquivo Box14.h
2  #define BOX14_H
3
4  class Box {
5  private:
6      double length, width, height;
7
8  public:
9      Box(double lv = 1.0, double wv = 1.0, double hv = 1.0)
10         : length{lv}, width{wv}, height{hv}
11     {}
12
13     double volume() { return length * width * height; }
14
15     double getLength() { return length; }
16     double getWidth() { return width; }
17     double getHeight() { return height; }
```

Ponteiro `this` — Exemplo 3 (cont.)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Retornando uma referência nos setters

```
18      Box& setLength(double length) {
19          this->length = length;
20          return *this;
21      }
22
23      Box& setWidth(double width) {
24          this->width = width;
25          return *this;
26      }
27
28      Box& setHeight(double height) {
29          this->height = height;
30          return *this;
31      }
32  };
33
34  #endif
```

Ponteiro `this` — Exemplo 3 (cont.)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Retornando uma referência nos setters

```
1  // arquivo mainBox14.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box14.h"
5  using namespace std;
6
7  int main() {
8      Box box;
9
10     // Atencao: agora eh possivel encadear chamadas
11     box.setLength(2.0).setWidth(3.0).setHeight(4.0);
12
13     cout << fixed << setprecision(2);
14     cout << "Length: " << box.getLength() << endl;
15     cout << "Width: " << box.getWidth() << endl;
16     cout << "Height: " << box.getHeight() << endl;
17     cout << "Volume: " << box.volume() << endl;
18 }
```

Objetos e Funções-membros constantes

Exemplo


```
1  #ifndef BOX15_H // Arquivo Box15.h
2  #define BOX15_H
3
4  class Box {
5  private:
6      double length, width, height;
7
8  public:
9      Box(double length=1, double width=1, double height=1)
10         : length{length}, width{width}, height{height}
11     {}
12
13     double volume() const {
14         return length * width * height;
15     }
16
17     double getLength() const { return length; }
18     double getWidth() const { return width; }
19     double getHeight() const { return height; }
```

Exemplo (cont.)


```
1       Box& setLength(double length) {
2           this->length = length;
3           return *this;
4       }
5
6       Box& setWidth(double width) {
7           this->width = width;
8           return *this;
9       }
10
11      Box& setHeight(double height) {
12          this->height = height;
13          return *this;
14      }
15  };
16
17  #endif
```

Exemplo (cont.)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // arquivo mainBox15.cpp
2  #include <iostream>
3  #include <iomanip>
4  #include "Box15.h"
5  using namespace std;
6
7  void print_box(const Box& box) {
8      cout << fixed << setprecision(2);
9      cout << "Length: " << box.getLength() << endl;
10     cout << "Width: " << box.getWidth() << endl;
11     cout << "Height: " << box.getHeight() << endl;
12     cout << "Volume: " << box.volume() << endl;
13 }
14
15 int main() {
16     print_box( Box{2.0, 3.0, 4.0} ); // imprime objeto anonimo
17 }
```

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM