TAD - Tipos Abstratos de Dados

Estrutura de Dados — QXD0010

Roberto Cabral
rbcabral@ufc.br

Universidade Federal do Ceará

2º semestre/2022

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.
  - **TAD** = dados + operações

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.
  - **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 Dado (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.

Tipos Abstratos de Dados (TADs)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

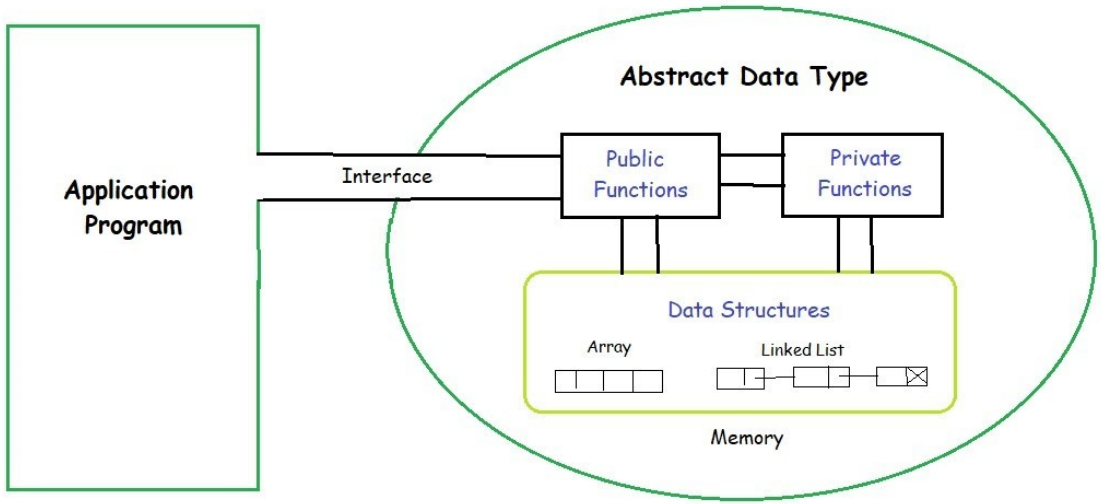

Imagem extraída de: `www.geeksforgeeks.org`

- A interface do TAD lista quais operações podem ser executadas, mas não como essas operações são implementadas.

Tipos Abstratos de Dados (TADs)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

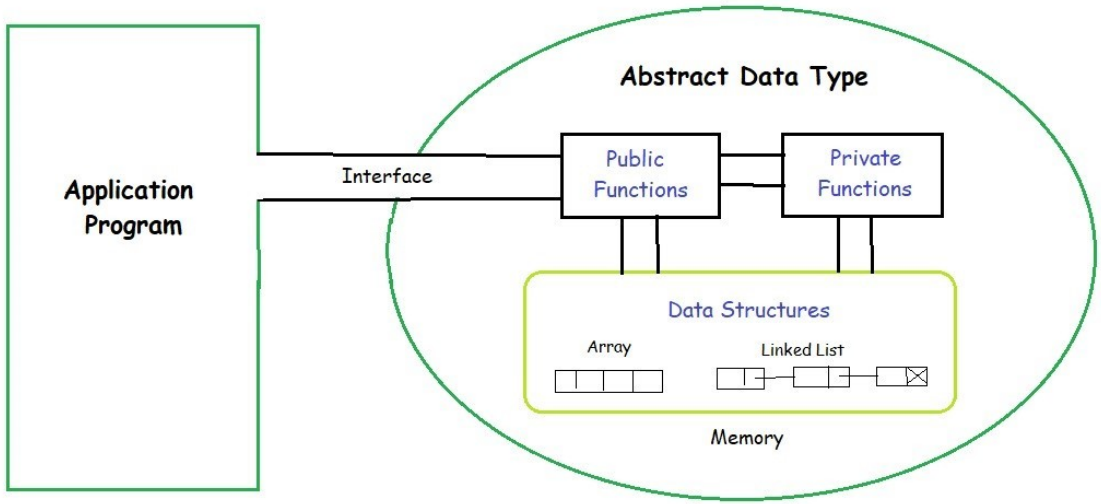

Imagem extraída de: `www.geeksforgeeks.org`

- A interface do TAD lista quais operações 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 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:

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 (em programação estruturada)

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 (em programação estruturada)
  - criação de classes (em programação orientada a objetos)

TAD Ponto

TAD Ponto

- 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.

TAD Ponto

- 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.
- Um ponto no $\mathbb{R}^2$ tem coordenadas x e y.
- 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`.
- Vou mostrar como implementar o TAD Ponto usando inicialmente programação estruturada. Depois, vou mostrar como implementar o TAD Ponto usando o paradigma de programação orientada a objetos.

Interface do TAD Ponto – Arquivo `Ponto.h`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1 #ifndef PONTO_H
2 #define PONTO_H
3
4 struct Ponto; // Tipo exportado
```

Interface do TAD Ponto – Arquivo `Ponto.h`


```
1 #ifndef PONTO_H
2 #define PONTO_H
3
4 struct Ponto; // Tipo exportado
5
6 // Aloca e retorna um ponto com coordenadas (x,y)
7 Ponto* pto_cria(double x, double y);
```

Interface do TAD Ponto – Arquivo `Ponto.h`


```
1  #ifndef PONTO_H
2  #define PONTO_H
3
4  struct Ponto; // Tipo exportado
5
6  // Aloca e retorna um ponto com coordenadas (x,y)
7  Ponto* pto_cria(double x, double y);
8
9  // Libera a memoria de um ponto previamente criado
10 void pto_libera(Ponto* p);
```

Interface do TAD Ponto – Arquivo Ponto.h


```
1  #ifndef PONTO_H
2  #define PONTO_H
3
4  struct Ponto; // Tipo exportado
5
6  // Aloca e retorna um ponto com coordenadas (x,y)
7  Ponto* pto_cria(double x, double y);
8
9  // Libera a memoria de um ponto previamente criado
10 void pto_libera(Ponto* p);
11
12 // Retorna as coordenadas de um ponto
13 double pto_getX(Ponto *p);
14 double pto_getY(Ponto *p);
```

Interface do TAD Ponto – Arquivo `Ponto.h`


```
1  #ifndef PONTO_H
2  #define PONTO_H
3
4  struct Ponto; // Tipo exportado
5
6  // Aloca e retorna um ponto com coordenadas (x,y)
7  Ponto* pto_cria(double x, double y);
8
9  // Libera a memoria de um ponto previamente criado
10 void pto_libera(Ponto* p);
11
12 // Retorna as coordenadas de um ponto
13 double pto_getX(Ponto *p);
14 double pto_getY(Ponto *p);
15
16 // Atribui novos valores as coordenadas de um ponto
17 void pto_setX(Ponto *p, double x);
18 void pto_setY(Ponto *p, double y);
```

Interface do TAD Ponto – Arquivo `Ponto.h`


```
1  #ifndef PONTO_H
2  #define PONTO_H
3
4  struct Ponto; // Tipo exportado
5
6  // Aloca e retorna um ponto com coordenadas (x,y)
7  Ponto* pto_cria(double x, double y);
8
9  // Libera a memoria de um ponto previamente criado
10 void pto_libera(Ponto* p);
11
12 // Retorna as coordenadas de um ponto
13 double pto_getX(Ponto *p);
14 double pto_getY(Ponto *p);
15
16 // Atribui novos valores as coordenadas de um ponto
17 void pto_setX(Ponto *p, double x);
18 void pto_setY(Ponto *p, double y);
19
20 // Retorna a distancia entre dois pontos
21 double pto_distancia(Ponto* p1, Ponto* p2);
22
23 #endif
```

Observações

- Os programas que quiserem utilizar esse TAD só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Ponto.h

Observações

- Os programas que quiserem utilizar esse TAD só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Ponto.h

- O arquivo de implementação do módulo (o arquivo Ponto.cpp) deve sempre incluir o arquivo de interface do módulo.

Observações

- Os programas que quiserem utilizar esse TAD só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Ponto.h

- O arquivo de implementação do módulo (o arquivo Ponto.cpp) deve sempre incluir o arquivo de interface do módulo.

- Isto é necessário por duas razões:

Observações

- Os programas que quiserem utilizar esse TAD só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Ponto.h

- O arquivo de implementação do módulo (o arquivo Ponto.cpp) deve sempre incluir o arquivo de interface do módulo.

- Isto é necessário por duas razões:
  - Podem existir definições na interface que são necessárias na implementação (isso não acontece no exemplo do TAD Ponto).

Observações

- Os programas que quiserem utilizar esse TAD só terão acesso às informações que possam ser obtidas através das funções exportadas pelo arquivo Ponto.h

- O arquivo de implementação do módulo (o arquivo Ponto.cpp) deve sempre incluir o arquivo de interface do módulo.

- Isto é necessário por duas razões:
  - Podem existir definições na interface que são necessárias na implementação (isso não acontece no exemplo do TAD Ponto).
  - Precisamos garantir que as funções implementadas correspondem às funções da interface. Como o protótipo das funções exportadas é incluído, o compilador verifica, por exemplo, se os parâmetros das funções implementadas equivalem aos parâmetros dos protótipos.

Usando a interface Ponto.h

- Se conhecermos apenas a interface do TAD, podemos criar programas que usem as funcionalidades exportadas.

- O arquivo que usa o TAD deve, obrigatoriamente, incluir o arquivo de cabeçalho responsável por definir sua interface.

Programa principal `mainPonto.cpp`

Programa principal mainPonto.cpp

Universidade Federal do Ceará
Campus Quixadá

```cpp
#include <iostream> // mainPonto.cpp
#include "Ponto.h"
using namespace std;

int main() {
    Ponto *p = pto_cria(2.0, 1.0);
    Ponto *q = pto_cria(3.4, 2.1);

    double d = pto_distancia(p, q);

    cout << "Distancia entre pontos: " << d << endl;

    pto_libera(p);
    pto_libera(q);

    return 0;
}
```

Implementação do TAD Ponto — `Ponto.cpp`

Implementação do TAD Ponto — Ponto.cpp

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Arquivo Ponto.cpp
2  // Implementacao do TAD Ponto
3  #include <iostream>
4  #include <cmath>
5  #include "Ponto.h"
6
7  struct Ponto {
8      double x;
9      double y;
10 };
```

Implementação do TAD Ponto — Ponto.cpp

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Arquivo Ponto.cpp
2  // Implementacao do TAD Ponto
3  #include <iostream>
4  #include <cmath>
5  #include "Ponto.h"
6
7  struct Ponto {
8      double x;
9      double y;
10 };
11
12 Ponto *pto_cria(double x, double y) {
```

Implementação do TAD Ponto — Ponto.cpp

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Arquivo Ponto.cpp
2  // Implementacao do TAD Ponto
3  #include <iostream>
4  #include <cmath>
5  #include "Ponto.h"
6
7  struct Ponto {
8      double x;
9      double y;
10 };
11
12 Ponto *pto_cria(double x, double y) {
13     Ponto *p = new (std::nothrow) Ponto;
14     if(p == nullptr) {
15         std::cerr << "Memoria insuficiente\n";
16         return nullptr;
17     }
18     p->x = x;
19     p->y = y;
20     return p;
21 }
```

Final do arquivo Ponto.cpp


```
22 // Libera a memoria de um ponto previamente criado
23 void pto_libera(Ponto *p) {
```

Final do arquivo Ponto.cpp


```
42 // Libera a memoria de um ponto previamente criado
43 void pto_libera(Ponto *p) {
44     if(p != nullptr) delete p;
45 }
```

Final do arquivo Ponto.cpp


```
62 // Libera a memoria de um ponto previamente criado
63 void pto_libera(Ponto *p) {
64     if(p != nullptr) delete p;
65 }
66
67 // Retorna os valores das coordenadas de um ponto
68 double pto_getX(Ponto *p) {
69     return p->x;
70 }
71 double pto_getY(Ponto *p) {
72     return p->y;
73 }
```

Final do arquivo Ponto.cpp


```
82 // Libera a memoria de um ponto previamente criado
83 void pto_libera(Ponto *p) {
84     if(p != nullptr) delete p;
85 }
86
87 // Retorna os valores das coordenadas de um ponto
88 double pto_getX(Ponto *p) {
89     return p->x;
90 }
91 double pto_getY(Ponto *p) {
92     return p->y;
93 }
94
95 // Atribui novos valores as coordenadas de um ponto
96 void pto_setX(Ponto *p, double x) {
97     p->x = x;
98 }
99 void pto_setY(Ponto *p, double y) {
100     p->y = y;
101 }
```

Final do arquivo Ponto.cpp


```
102 // Libera a memoria de um ponto previamente criado
103 void pto_libera(Ponto *p) {
104     if(p != nullptr) delete p;
105 }
106
107 // Retorna os valores das coordenadas de um ponto
108 double pto_getX(Ponto *p) {
109     return p->x;
110 }
111 double pto_getY(Ponto *p) {
112     return p->y;
113 }
114
115 // Atribui novos valores as coordenadas de um ponto
116 void pto_setX(Ponto *p, double x) {
117     p->x = x;
118 }
119 void pto_setY(Ponto *p, double y) {
120     p->y = y;
121 }
```

Final do arquivo Ponto.cpp


```
122 // Retorna a distancia entre dois pontos
123 double pto_distancia(Ponto* p1, Ponto* p2) {
```

Final do arquivo Ponto.cpp


```
128 // Retorna a distancia entre dois pontos
129 double pto_distancia(Ponto* p1, Ponto* p2) {
130     double dx = p2->x - p1->x;
131     double dy = p2->y - p1->y;
132     return sqrt(dx*dx + dy*dy);
133 }
```

Compilação do Projeto

- Note que o projeto tem dois arquivos de implementação, o arquivo mainPonto.cpp e o arquivo Ponto.cpp. Somente eles devem ser compilados. O arquivo de cabeçalho não deve ser compilado.

- Para compilar o projeto por linha de comando:

  g++ *.cpp -o main

- Para executar:

  ./main

Exercício

Exercício — TAD Círculo

Vamos considerar a criação de um TAD para representar um círculo no $\mathbb{R}^2$.

- Implemente o TAD Círculo usando programação estruturada por meio de módulos (como foi feito para o TAD Ponto).
- O Círculo é definido pelo seu **ponto central** e pelo seu **raio**.
  O seu TAD deve ter as seguintes funções:
  - `Circulo *circ_cria(double raio, Ponto centro)`: cria um círculo cujo centro é um atributo do tipo `Ponto` e raio é um `double`.
  - `void circ_setRaio(Circulo *c, double r)`: atribui novo valor ao raio do círculo.
  - `double circ_getRaio(Circulo *c)` obtém o raio.
  - `Ponto *circ_getCentro(Circulo *c)`: obtém o centro.
  - `double circ_area(Circulo *c)`: calcula e retorna a área do círculo.
  - `bool circ_interior(Circulo *c, Ponto p)`: verifica se o Ponto p está dentro do círculo.
  - `void circ_libera(Circulo *c)`: libera a memória alocada para c.

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

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, 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.

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).

- 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.

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

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

Os modificadores de acesso que usaremos são `public` e `private`.

Modificadores de acesso

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

Os modificadores de acesso que usaremos são `public` e `private`.

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

Modificadores de acesso

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

Os modificadores de acesso que usaremos 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.

Construtor e Destrutor

- Todas as classes em C++ possuem 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> // construtor4.cpp
2
3  class Ponto { // tridimensional
4  private:
5      double x, y, z;
6  public:
7      // permite alguns argumentos nao serem fornecidos
8      Ponto(double X = 0, double Y = 0, double Z = 0) {
9          x = X;
10         y = Y;
11         z = Z;
12         std::cout << "(" << x << "," << y << "," << z << ")";
13     }
14 };
15
16 int main() {
17     Ponto p1 { 4, 5, 7 };
18     Ponto p2 { 4, 5 };
19     Ponto p3 { 4 };
20     Ponto p4;
21 }
```

Implementando um construtor (3)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

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

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.

Destrutor

- Destrutor é uma função-membro especial que é sempre invocada quando o objeto é liberado.
- O destrutor serve para 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> // destrutor.cpp
2
3  class Ponto {
4  private:
5      double x;
6      double y;
7
8  public:
9      Ponto(double X, double Y) { x = X; y = Y; } // construtor
10
11     // Destrutor (note o til antes do nome da funcao)
12     ~Ponto() {
13         std::cout << "Ponto destruido\n";
14     }
15
16     double getX() { return x; } // getters
17     double getY() { return y; } // getters
18
19     void setX(double newX) { x = newX; } // setters
20     void setY(double newY) { y = newY; } // setters
21 };
```

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

Relembrando a interface

- Criar de um TAD para representar um ponto no espaço $\mathbb{R}^2$.

- Para isso, devemos definir um tipo abstrato, que denominamos Ponto, os seus atributos e o conjunto de funções-membro que operam sobre esse tipo.

Relembrando a interface

- Criar de um TAD para representar um ponto no espaço $\mathbb{R}^2$.

- Para isso, devemos definir um tipo abstrato, que denominamos Ponto, os seus atributos e o conjunto de funções-membro que operam sobre esse tipo.

- Neste exemplo, vamos considerar as seguintes operações:
  - `cria`: cria um ponto com coordenadas x e y
  - `libera`: se for necessário, libera a memória alocada por um ponto
  - `acessa x`: devolve a coordenada x de um ponto
  - `acessa y`: devolve a coordenada y de um ponto
  - `atribui x`: atribui novo valor à coordenada x do ponto
  - `atribui y`: atribui novo valor à coordenada y do ponto
  - `distancia`: calcula a distância entre dois pontos.

Arquivo Ponto2.h


```
1  #ifndef PONTO_H
2  #define PONTO_H
3  #include <cmath>
4  #include <string>
5  #include <sstream>
6  #include <iostream>
7
8  class Ponto {
9  private:
10     double x, y;
11 public:
12     // Construtores
13     Ponto(double x, double y = 0) {
14         this->x = x;
15         this->y = y;
16     }
17
18     Ponto() : Ponto(0,0) { }
19
20     // Destrutor
21     ~Ponto() {
22         std::cout << toString() << " liberado\n";
23     }
```

Arquivo Ponto2.h

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
24      // Getters
25      double getX() { return x; }
26      double getY() { return y; }
27
28      // Setters
29      void setX(double x) { this->x = x; }
30      void setY(double y) { this->y = y; }
31
32      // Calcula a distancia entre dois pontos:
33      // Entre o ponto que chamou essa funcao
34      // e o ponto p passado como parametro
35      double distancia(Ponto *p) {
36          double dx = pow(this->x - p->x, 2);
37          double dy = pow(this->y - p->x, 2);
38          return sqrt(dx + dy);
39      }
```

Arquivo Ponto2.h

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
40      // retorna o ponto como uma string
41      std::string toString() {
42          std::stringstream sx, sy;
43          sx << x;
44          sy << y;
45          return "(" + sx.str() + "," + sy.str() + ")";
46      }
47  };
48
49  #endif
```

Programa Cliente — main2.cpp

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #include <iostream> // main2.cpp
2  #include "Ponto3.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: " << p1.toString() << endl;
11
12     cout << "Ponto 2: " << p2.toString() << endl;
13
14     cout << "Ponto 3: " << p3.toString() << endl;
15
16     cout << "Distancia: " << p1.distancia(&p2) << endl;
17     return 0;
18 }
```

Outra Implementação do TAD Ponto

Arquivo Ponto3.h


```
1  #ifndef PONTO3_H
2  #define PONTO3_H
3
4  struct Ponto {
5  private:
6      double x, y;
7  public:
8      Ponto();
9      Ponto(double X, double Y);
10
11     ~Ponto();
12
13     double getX();
14     double getY();
15
16     void setX(double x);
17     void setY(double y);
18
19     double distancia(Ponto *p);
20 };
21
22 #endif
```

Arquivo Ponto3.cpp


```cpp
#include <iostream>
#include <cmath>
#include "Ponto3.h"

Ponto::Ponto() {
    x = y = 0;
}

Ponto::Ponto(double x, double y) {
    this->x = x;
    this->y = y;
}

Ponto::~Ponto() {
    std::cout << "Ponto destruido" << std::endl;
}
```

Final do Arquivo Ponto3.cpp


```
18 double Ponto::getY() { return y; }
19
20 void Ponto::setX(double x) { this->x = x; }
21 void Ponto::setY(double y) { this->y = y; }
22
23 double Ponto::distancia(Ponto *p) {
24     double dx = pow(x - p->x, 2);
25     double dy = pow(y - p->y, 2);
26     return sqrt(dx + dy);
27 }
```

Referências em C++

Referências

- Frequentemente precisamos referenciar um objeto
  - sem fazer uma cópia do objeto

- Há dois modos de fazermos isso:
  - **Indiretamente**, por meio de um ponteiro
    - dá o endereço (em memória) do objeto
    - Requer o uso de trabalho extra: derreferenciação
  - **Diretamente**, por meio de uma referência
    - age como um alias(apelido) para o objeto
    - O usuário interage com a referência como se ela fosse o próprio objeto.

Referências – Exemplo 1

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
#include <iostream> // Referencia01.cpp
using namespace std;

int main() {
    int x = 45;

    int& ref = x; // criação de uma referência

    cout << ref << endl; // posso usar ref no lugar de x

    ref = 67; // muda o valor de x para 67

    cout << x << endl; // x mudou de valor ----> 67
}
```

Referências – Exemplo 2


```
1  #include <iostream> // Referencia02.cpp
2  using namespace std;
3
4  void troca(int& x, int& y) {
5      int aux = x;
6      x = y;
7      y = aux;
8  }
9
10 int main() {
11     int a = 45;
12     int b = 67;
13
14     troca(a,b);
15
16     cout << "a: " << a << endl; // imprime 67
17     cout << "b: " << b << endl; // imprime 45
18 }
```

Referências – Exemplo 3


```cpp
#include <iostream> // Referencia03.cpp
using namespace std;

struct Ponto {
    double x = 0, y = 0;
};

void lerPonto(Ponto& p) {
    cin >> p.x;
    cin >> p.y;
}

int main() {
    Ponto ponto;
    lerPonto(ponto);
    cout << ponto.x << "," << ponto.y << endl;
}
```

O que é uma referência em C++?

- Uma variável que guarda um endereço
- Porém com uma interface mais amigável que um ponteiro
  - Uma referência para um objeto oculta a indireção do programador.
- Referências devem ser tipadas
  - checadas pelo compilador
  - assim como ponteiros, elas só podem referenciar o tipo para o qual elas podem apontar.
- Referências devem obrigatoriamente referenciar alguma coisa.
  - devem ser inicializadas

Referências vs Ponteiros

- Depois que uma referência é criada, ela não pode referenciar outro objeto. Já com ponteiros isso é possível.

- Referências não podem ser `null`, enquanto ponteiros podem. Toda referência deve referenciar algum objeto.
  - Por esse motivo, não podemos ter um array de referências por exemplo, já que referências devem ser inicializadas no momento em que são declaradas.

- Não é possível referenciar diretamente um objeto do tipo referência depois que ele é definido. Qualquer ocorrência do seu nome refere-se diretamente ao objeto que ele referencia.

Quando usar referência?

- Motivo 1: evitar fazer uma cópia de objetos ou structs muito grandes ao passá-los como argumentos para funções.

- Motivo 2: quando você quiser modificar o valor do parâmetro de entrada de uma função e não houver a necessidade do uso de ponteiros para fazer isso.

Quando não usar referência?

Universidade Federal do Ceará
Campus Quixadá

Funções não devem retornar uma referência para variáveis locais.

```
1  #include <iostream> // Referencia04.cpp
2  using namespace std;
3
4  // Código inválido
5  int& getLocalVariable() {
6      int x = new int;
7      x = 45;
8      return x;
9  }
10
11 int main() {
12     cout << getLocalVariable() << endl;
13 }
```

Quando não usar referência?

Funções não devem retornar uma referência para variáveis locais.

```
1  #include <iostream> // Referencia04.cpp
2  using namespace std;
3
4  // Código inválido
5  int& getLocalVariable() {
6      int x = new int;
7      x = 45;
8      return x;
9  }
10
11 int main() {
12     cout << getLocalVariable() << endl;
13 }
```

- Como x é local, ela é destruída logo depois da função terminar. Logo, ela não existe mais quando a função main executar.

Referências e a palavra-chave const

Referências para valores constantes

- É possível declarar uma referência para um valor constante. Isso é feito declarando a referência com a palavra-chave `const`

```
const int apples = 5;
const int& ref = apples;
```

Referências para valores constantes

- É possível declarar uma referência para um valor constante. Isso é feito declarando a referência com a palavra-chave `const`

```
const int apples = 5;
const int& ref = apples;
```

- **Atenção:** Referências para valores não-constantes só podem ser iniciadas com dados(variáveis/valores) não-constantes.

```
const int apples = 5;
int& ref = apples; // erro
```

Iniciando referências para valores constantes

- Referências para valores constantes podem ser iniciadas com:
  - variáveis não-constantes
  - variáveis constantes
  - valores temporários (literais, constantes, objetos anônimos)

```
int x = 5;
const int& ref1 = x; // okay, x é um valor não-const

const int y = 7;
const int& ref2 = y; // okay, y é um valor const

const int& ref3 = 6; // okay, 6 é uma constante literal
```

Iniciando referências para valores constantes

- Referências para valores constantes podem ser iniciadas com:
  - variáveis não-constantes
  - variáveis constantes
  - valores temporários (literais, constantes, objetos anônimos)

```
int x = 5;
const int& ref1 = x; // okay, x é um valor não-const

const int y = 7;
const int& ref2 = y; // okay, y é um valor const

const int& ref3 = 6; // okay, 6 é uma constante literal
```

- Quando acessado a partir de uma referência para valor constante, um valor é considerado `const` mesmo se a variável original não for `const`.

Referências para valores temporários

- Referências para valores temporários estendem o tempo de vida do valor referenciado.
  - Geralmente, os valores temporários são destruídos ao final da expressão em que eles são criados.
  - Exemplo:

```
cout << 2+3; //2+3 é avaliado para 5, que é destruído ao final da declaração
```

Referências para valores temporários

- Referências para valores temporários estendem o tempo de vida do valor referenciado.
  - Geralmente, os valores temporários são destruídos ao final da expressão em que eles são criados.
  - Exemplo:

```
cout << 2+3; //2+3 é avaliado para 5, que é destruído ao
final da declaração
```

- Contudo, quando uma referência para valor constante é iniciada com um valor temporário, o tempo de vida do valor temporário é estendido para o tempo de vida da referência. Exemplo:

```
int func() {
  const int& ref = 2+3; //normalmente o resultado de 2+3 é
  //destruído ao final desta expressão
  cout << ref; //porém, como ele foi atribuído a uma
  //referência, o seu tempo de vida é estendido
  //até aqui, quando a referência morre
}
```

Exemplo


```
1  #include <iostream> // Referencia05.cpp
2  using namespace std;
3
4  int soma1 (int& x, int& y) { // non-const
5      return x + y;
6  }
7
8  int soma2 (const int& x, const int& y) {
9      return x + y;
10 }
11
12 int main() {
13     int a = 5;
14     int b = 6;
15
16     cout << soma1(a, b) << endl;
17     cout << soma2(a, b) << endl;
18     cout << soma1(3, 4) << endl; // erro
19     cout << soma2(3, 4) << endl;
20 }
```

Exercícios

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 os seguintes métodos:
  - 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(Ponto& p)`: muda o centro.
  - `double getRadius()` obtém o raio.
  - `Ponto getCenter()`: obtém o centro.
  - `double area()`: retorna a área do círculo.
  - `bool interior(Ponto& p)`: verifica se p está dentro do círculo.

Exercício — TAD Matrix

- Criar um TAD para representar uma 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.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM