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/2021

Introdução

Tipos Abstratos de Dados (TADs)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- 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)

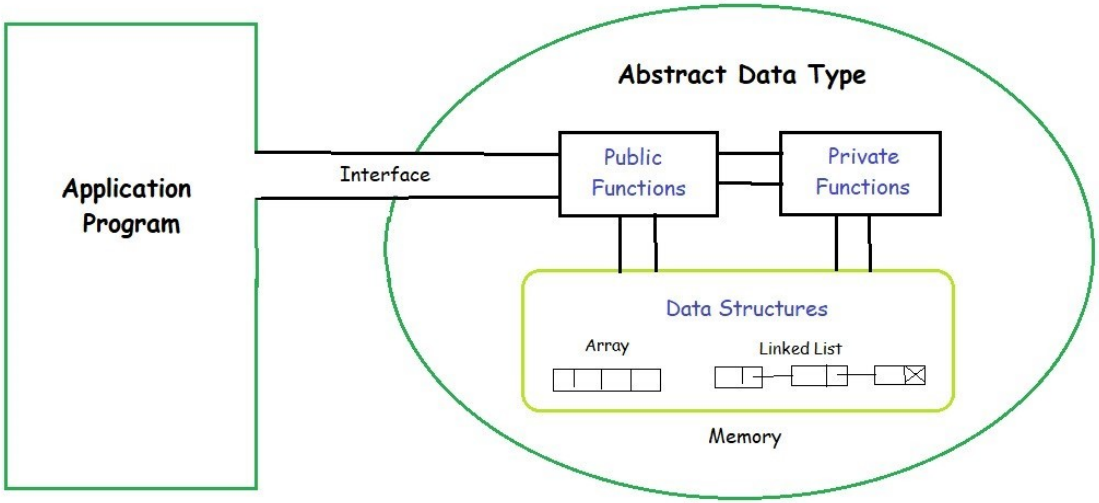


Imagem extraída de: `www.geeksforgeeks.org`

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

Tipos Abstratos de Dados (TADs)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

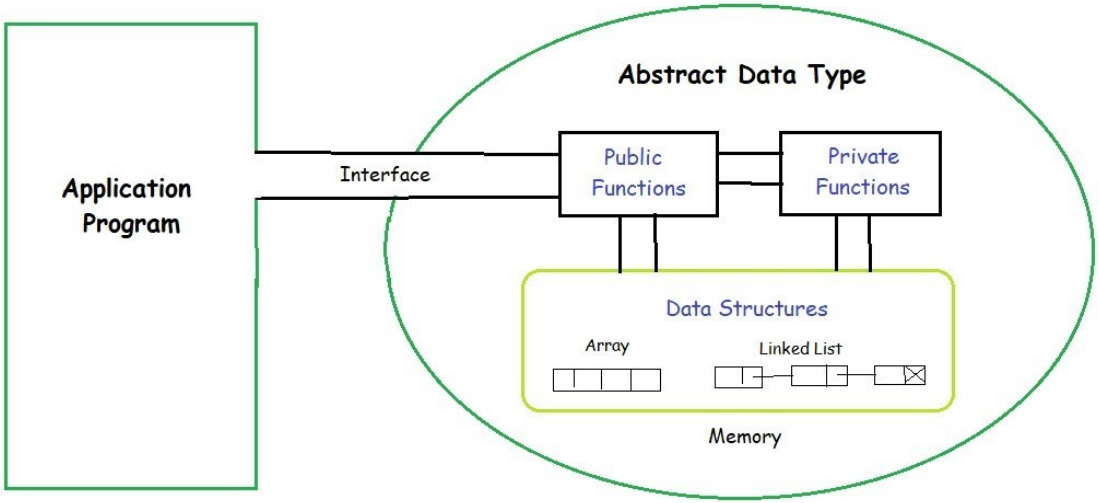


Imagem extraída de: `www.geeksforgeeks.org`

- 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)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

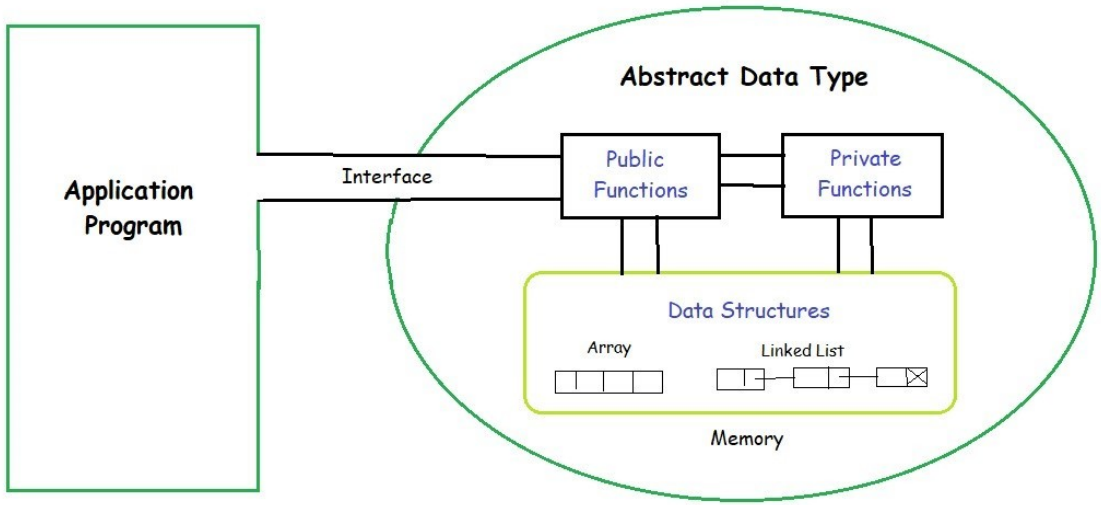


Imagem extraída de: `www.geeksforgeeks.org`

- 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

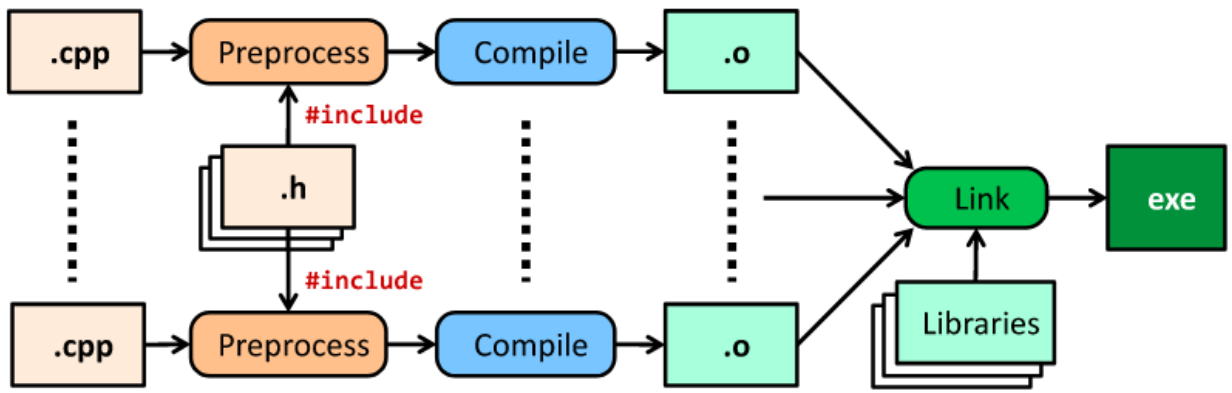


Imagem extraída de: `https://hackingcpp.com/cpp/lang/separate_compilation.html`

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`.
- 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 // Arquivo Ponto.h --- Interface do TAD Ponto
4
5 struct Ponto; // Tipo exportado
```

Interface do TAD Ponto – Arquivo `Ponto.h`


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

Interface do TAD Ponto – Arquivo `Ponto.h`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

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

Interface do TAD Ponto – Arquivo Ponto.h

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #ifndef PONTO_H
2  #define PONTO_H
3  // Arquivo Ponto.h --- Interface do TAD Ponto
4
5  struct Ponto; // Tipo exportado
6
7  // Aloca e retorna um ponto com coordenadas (x,y)
8  Ponto *pto_cria(double x, double y);
9
10 // Libera a memoria de um ponto previamente criado
11 void pto_libera(Ponto* p);
12
13 // Retorna os valores das coordenadas de um ponto
14 // nos parametros x e y
15 void pto_acessa(Ponto *p, double* x, double *y);
```

Interface do TAD Ponto – Arquivo Ponto.h

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #ifndef PONTO_H
2  #define PONTO_H
3  // Arquivo Ponto.h --- Interface do TAD Ponto
4
5  struct Ponto; // Tipo exportado
6
7  // Aloca e retorna um ponto com coordenadas (x,y)
8  Ponto *pto_cria(double x, double y);
9
10 // Libera a memoria de um ponto previamente criado
11 void pto_libera(Ponto* p);
12
13 // Retorna os valores das coordenadas de um ponto
14 // nos parametros x e y
15 void pto_acessa(Ponto *p, double* x, double *y);
16
17 // Atribui novos valores as coordenadas de um ponto
18 void pto_atribui(Ponto *p, double x, double y);
```

Interface do TAD Ponto – Arquivo Ponto.h

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  #ifndef PONTO_H
2  #define PONTO_H
3  // Arquivo Ponto.h --- Interface do TAD Ponto
4
5  struct Ponto; // Tipo exportado
6
7  // Aloca e retorna um ponto com coordenadas (x,y)
8  Ponto *pto_cria(double x, double y);
9
10 // Libera a memoria de um ponto previamente criado
11 void pto_libera(Ponto* p);
12
13 // Retorna os valores das coordenadas de um ponto
14 // nos parametros x e y
15 void pto_acessa(Ponto *p, double* x, double *y);
16
17 // Atribui novos valores as coordenadas de um ponto
18 void pto_atribui(Ponto *p, double x, 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.

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

```
1  #include <iostream> // mainPonto.cpp
2  #include "Ponto.h"
3  using namespace std;
4
5  int main() {
6      Ponto *p = pto_cria(2.0, 1.0);
7      Ponto *q = pto_cria(3.4, 2.1);
8
9      double d = pto_distancia(p, q);
10
11     cout << "Distancia entre pontos: " << d << endl;
12
13     pto_libera(p);
14     pto_libera(q);
15
16     return 0;
17 }
```

Implementação do TAD Ponto — `Ponto.cpp`

Implementação do TAD Ponto — Ponto.cpp

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
// Arquivo Ponto.cpp
// Implementacao do TAD Ponto
#include <iostream>
#include <cmath>
#include "Ponto.h"
using namespace std;

struct Ponto {
    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  using namespace std;
7
8  struct Ponto {
9      double x;
10     double y;
11 };
12
13 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  using namespace std;
7
8  struct Ponto {
9      double x;
10     double y;
11 };
12
13 Ponto *pto_cria(double x, double y) {
14     Ponto *p = new (nothrow) Ponto;
15     if(p == nullptr) {
16         cerr << "Memoria insuficiente" << endl;
17         return nullptr;
18     }
19     p->x = x;
20     p->y = y;
21     return p;
22 }
```

Final do arquivo Ponto.cpp


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

Final do arquivo Ponto.cpp


```
47 // Libera a memoria de um ponto previamente criado
48 void pto_libera(Ponto *p) {
49     if(p != nullptr) delete p;
50 }
```

Final do arquivo Ponto.cpp


```
70 // Libera a memoria de um ponto previamente criado
71 void pto_libera(Ponto *p) {
72     if(p != nullptr) delete p;
73 }
74
75 // Retorna os valores das coordenadas de um ponto
76 void pto_acessa(Ponto *p, double* x, double *y) {
```

Final do arquivo Ponto.cpp


```
93  // Libera a memoria de um ponto previamente criado
94  void pto_libera(Ponto *p) {
95      if(p != nullptr) delete p;
96  }
97
98  // Retorna os valores das coordenadas de um ponto
99  void pto_acessa(Ponto *p, double* x, double *y) {
100     *x = p->x;
101     *y = p->y;
102 }
```

Final do arquivo Ponto.cpp


```
116 // Libera a memoria de um ponto previamente criado
117 void pto_libera(Ponto *p) {
118     if(p != nullptr) delete p;
119 }
120
121 // Retorna os valores das coordenadas de um ponto
122 void pto_acessa(Ponto *p, double* x, double *y) {
123     *x = p->x;
124     *y = p->y;
125 }
126
127 // Atribui novos valores as coordenadas de um ponto
128 void pto_atribui(Ponto *p, double x, double y) {
```

Final do arquivo Ponto.cpp


```
139 // Libera a memoria de um ponto previamente criado
140 void pto_libera(Ponto *p) {
141     if(p != nullptr) delete p;
142 }
143
144 // Retorna os valores das coordenadas de um ponto
145 void pto_acessa(Ponto *p, double* x, double *y) {
146     *x = p->x;
147     *y = p->y;
148 }
149
150 // Atribui novos valores as coordenadas de um ponto
151 void pto_atribui(Ponto *p, double x, double y) {
152     p->x = x;
153     p->y = y;
154 }
```

Final do arquivo Ponto.cpp


```
162 // Libera a memoria de um ponto previamente criado
163 void pto_libera(Ponto *p) {
164     if(p != nullptr) delete p;
165 }
166
167 // Retorna os valores das coordenadas de um ponto
168 void pto_acessa(Ponto *p, double* x, double *y) {
169     *x = p->x;
170     *y = p->y;
171 }
172
173 // Atribui novos valores as coordenadas de um ponto
174 void pto_atribui(Ponto *p, double x, double y) {
175     p->x = x;
176     p->y = y;
177 }
178
179 // Retorna a distancia entre dois pontos
180 double pto_distancia(Ponto* p1, Ponto* p2) {
```

Final do arquivo Ponto.cpp


```
185 // Libera a memoria de um ponto previamente criado
186 void pto_libera(Ponto *p) {
187     if(p != nullptr) delete p;
188 }
189
190 // Retorna os valores das coordenadas de um ponto
191 void pto_acessa(Ponto *p, double* x, double *y) {
192     *x = p->x;
193     *y = p->y;
194 }
195
196 // Atribui novos valores as coordenadas de um ponto
197 void pto_atribui(Ponto *p, double x, double y) {
198     p->x = x;
199     p->y = y;
200 }
201
202 // Retorna a distancia entre dois pontos
203 double pto_distancia(Ponto* p1, Ponto* p2) {
204     double dx = p2->x - p1->x;
205     double dy = p2->y - p1->y;
206     return sqrt(dx*dx + dy*dy);
207 }
```

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

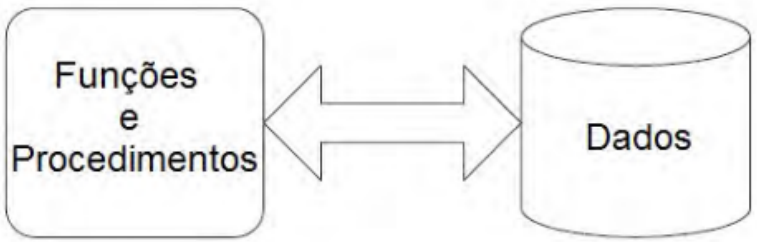
- Vamos considerar a criação de um tipo de dado para representar um círculo no $\mathbb{R}^2$.
- Implemente o TAD Círculo usando programação estruturada por meio de módulos. O seu módulo deve ter as seguintes funções:
  - Circulo *`circ_cria(float raio, Ponto centro)`: cria um círculo cujo centro é um atributo do tipo `Ponto` e raio é um `float`.
  - `void circ_setRaio(float r)`: atribui novo valor ao raio do círculo.
  - `float circ_getRaio()` obtém o raio.
  - `Ponto circ_getCentro()`: obtém o centro.
  - `float circ_area()`: calcula a área do círculo.
  - `bool circ_interior(Ponto p)`: verifica se o Ponto p está dentro do círculo.
  - `void circ_libera(Circulo *c)`: libera a memória alocada para c.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

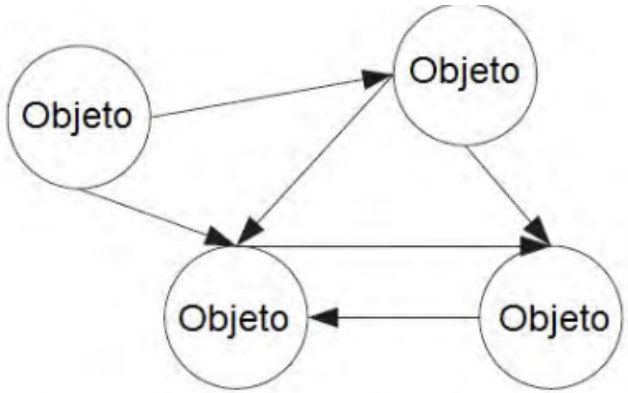
Programação orientada a objetos

Paradigma Orientado a Objetos

- A orientação a objetos é um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos.



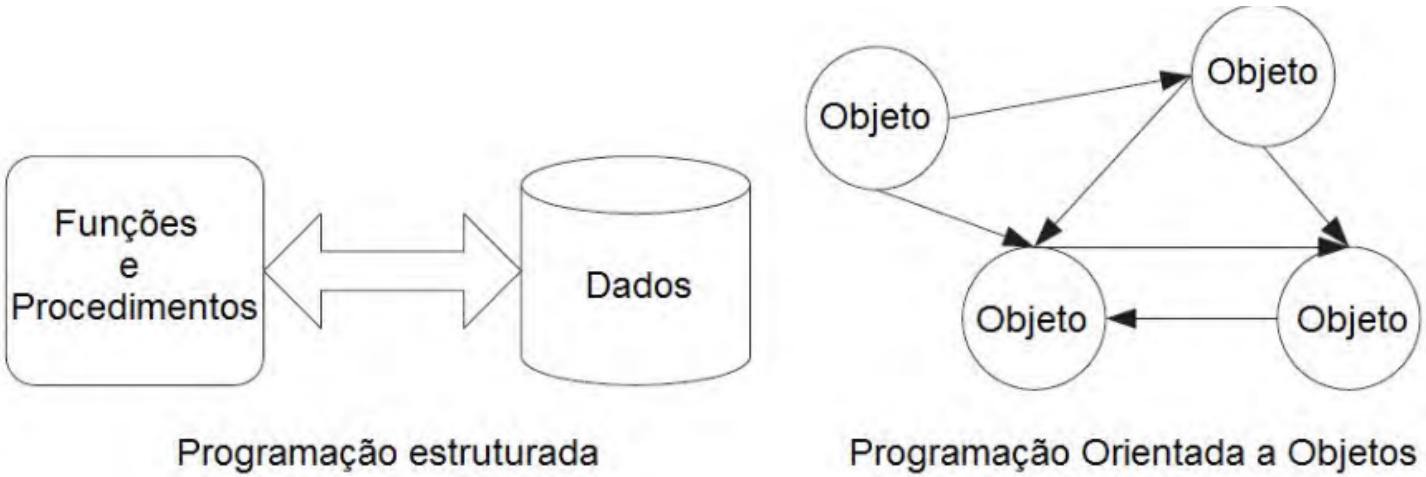
Programação estruturada



Programação Orientada a Objetos

Paradigma Orientado a Objetos

- A orientação a objetos é um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos.



- POO considera que os dados a serem processados e os mecanismos de processamento desses dados devem ser considerados em conjunto.
  - Precisamos de modelos que representem conjuntamente dados e operações nestes dados.

Modelos

O que são modelos?

Modelos são representações simplificadas de objetos, pessoas, itens, processos, conceitos, ideias, etc. usados comumente por pessoas no seu dia-a-dia, independente do uso de computadores.

O que são modelos?

Modelos são representações simplificadas de objetos, pessoas, itens, processos, conceitos, ideias, etc. usados comumente por pessoas no seu dia-a-dia, independente do uso de computadores.

Exemplo: Considere o **Restaurante Caseiro Dona Maria**, que serve refeições por quilo, e onde o gerente, que também é a pessoa que fica na balança e no caixa, anota os pesos dos pratos dos clientes e os pedidos que os garçons trazem em um quadro branco.

O que são modelos?

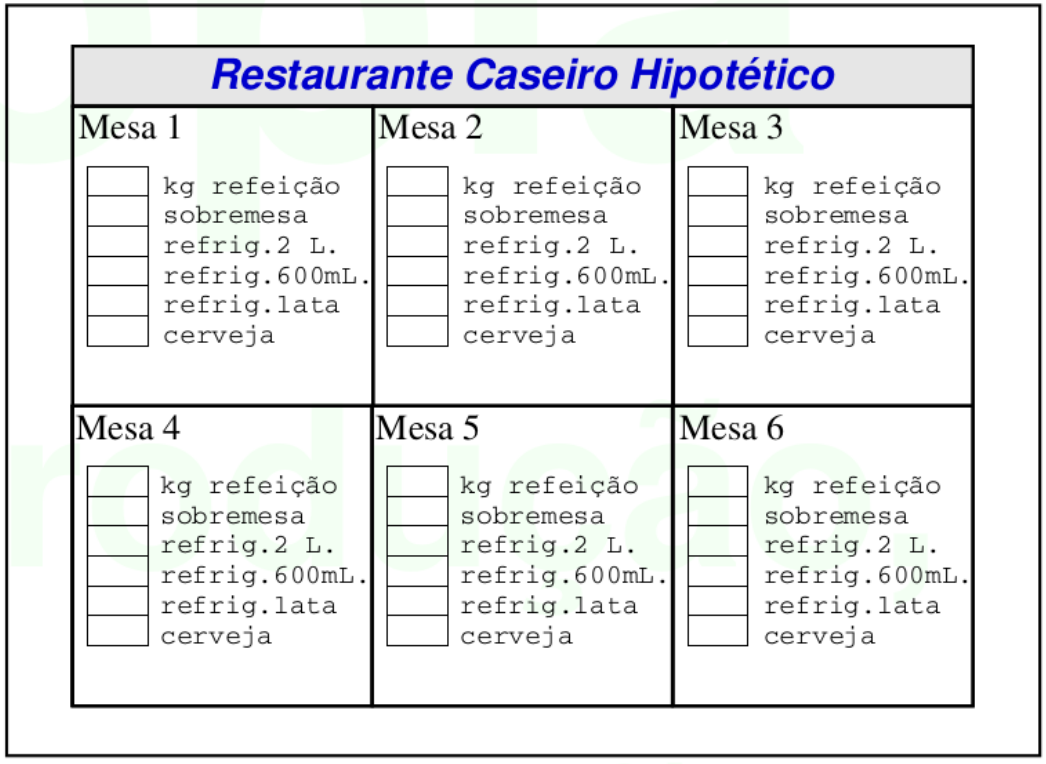
Modelos são representações simplificadas de objetos, pessoas, itens, processos, conceitos, ideias, etc. usados comumente por pessoas no seu dia-a-dia, independente do uso de computadores.

Exemplo: Considere o **Restaurante Caseiro Dona Maria**, que serve refeições por quilo, e onde o gerente, que também é a pessoa que fica na balança e no caixa, anota os pesos dos pratos dos clientes e os pedidos que os garçons trazem em um quadro branco.

Quando os itens dos pedidos são servidos, o gerente anota, ao lado do item no quadro-branco, o número de itens ou o peso do prato.

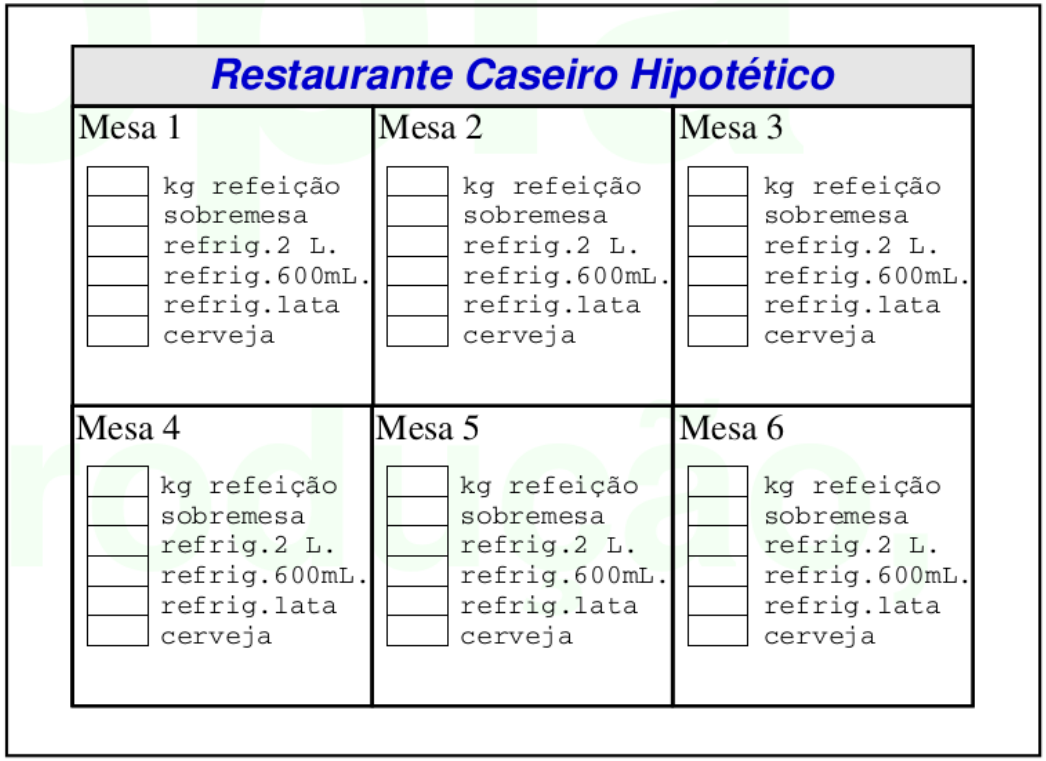
Quando o cliente pede a conta, o gerente se refere ao quadro-branco para calcular o valor devido.

Modelo do quadro-branco do Restaurante



Modelo do quadro-branco do Restaurante

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



- O quando-branco é um modelo do restaurante. Representa de forma simplificada as informações que são necessárias para a contabilização dos pedidos feitos pelos clientes.
  - Quais informações podemos extrair desse modelo?

Modelagem de dados e POO

- A criação e o uso de modelos é uma tarefa natural e a extensão desta abordagem à programação deu origem ao paradigma *Programação Orientada a Objetos*.

O que é a programação orientada a objetos?

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Programação orientada a objetos (POO) é um paradigma de programação de computadores onde se usam classes e objetos, criados a partir de modelos, para representar e processar dados usando programas de computador.

O que é a programação orientada a objetos?

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Programação orientada a objetos (POO) é um paradigma de programação de computadores onde se usam classes e objetos, criados a partir de modelos, para representar e processar dados usando programas de computador.

- Em POO, os dados pertencentes aos modelos são representados por tipos de dados nativos ou também podem ser representados por modelos já existentes na linguagem ou por outros modelos criados pelo programador.

O que é a programação orientada a objetos?

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Programação orientada a objetos (POO) é um paradigma de programação de computadores onde se usam classes e objetos, criados a partir de modelos, para representar e processar dados usando programas de computador.

- Em POO, os dados pertencentes aos modelos são representados por tipos de dados nativos ou também podem ser representados por modelos já existentes na linguagem ou por outros modelos criados pelo programador.

- Em POO, os dados e as operações que os manipulam são considerados em conjunto, como se fossem uma unidade.

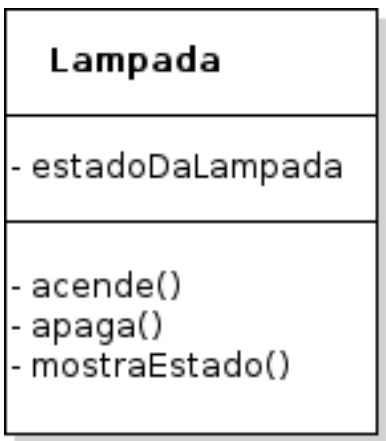
Mais exemplos de modelos

Lâmpada Incandescente

- Consideremos uma lâmpada incandescente que tem um dado básico, que é seu estado ("liga" ou "desligada").
- As operações que podem ser efetuadas na lâmpada são simples: podemos ligá-la ou desligá-la
- Para saber se uma lâmpada está ligada ou desligada podemos pedir que uma operação mostre o valor do estado.

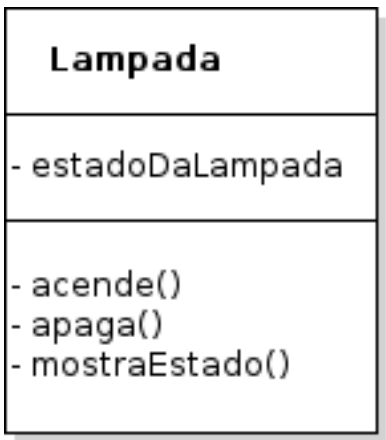
Lâmpada Incandescente

- Consideremos uma lâmpada incandescente que tem um dado básico, que é seu estado ("liga" ou "desligada").
- As operações que podem ser efetuadas na lâmpada são simples: podemos ligá-la ou desligá-la
- Para saber se uma lâmpada está ligada ou desligada podemos pedir que uma operação mostre o valor do estado.
- A figura abaixo mostra uma variante do diagrama de classes da Linguagem Unificada de Modelagem (*Unified Modeling Language*, UML)



Lâmpada Incandescente

- Consideremos uma lâmpada incandescente que tem um dado básico, que é seu estado ("liga" ou "desligada").
- As operações que podem ser efetuadas na lâmpada são simples: podemos ligá-la ou desligá-la
- Para saber se uma lâmpada está ligada ou desligada podemos pedir que uma operação mostre o valor do estado.
- A figura abaixo mostra uma variante do diagrama de classes da Linguagem Unificada de Modelagem (*Unified Modeling Language*, UML)



- Se as lâmpadas a serem representadas neste modelo fossem usadas em uma aplicação de controle de qualidade, quais dados seriam úteis?

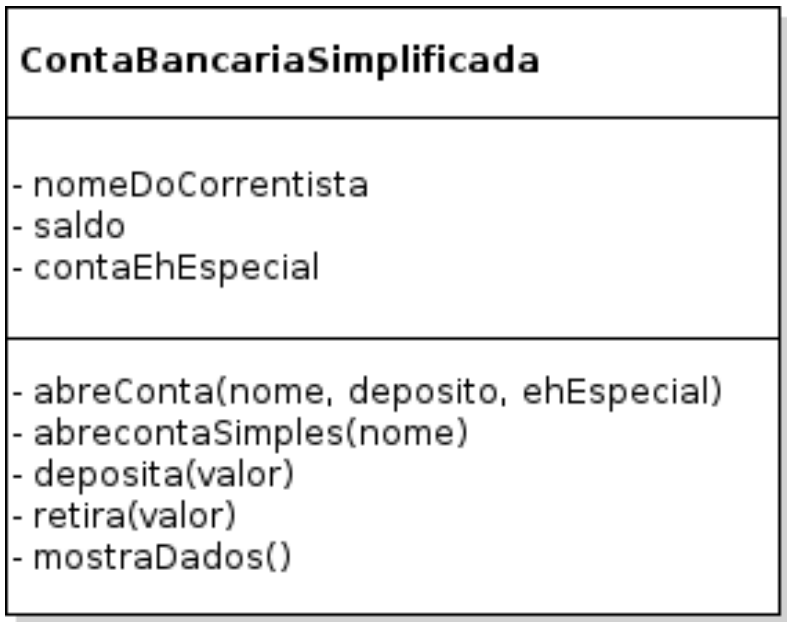
Lâmpada Incandescente

Pseudocódigo do modelo

```
1  modelo Lampada // representa uma lampada em uso
2  inicio do modelo
3    dado estadoDaLampada;
4
5    operacao acende() // acende a lampada
6      inicio
7        estadoDaLampada = aceso;
8      fim
9
10   operacao apaga() // apaga a lampada
11     inicio
12       estadoDaLampada = apagado;
13     fim
14
15   operacao mostraEstado() // mostra estado da lampada
16     inicio
17       se (estadoDaLampada == aceso)
18         imprime "A lampada esta acesa";
19       senao
20         imprime "A lampada esta apagada";
21     fim
22 fim do modelo
```

Uma conta bancária simplificada

- Este modelo de conta bancária somente representa o nome do correntista, o saldo da conta e se a conta é especial ou não
- Se a conta for especial, o correntista terá o direito de retirar mais dinheiro do que tem no saldo (ficar com o saldo negativo)



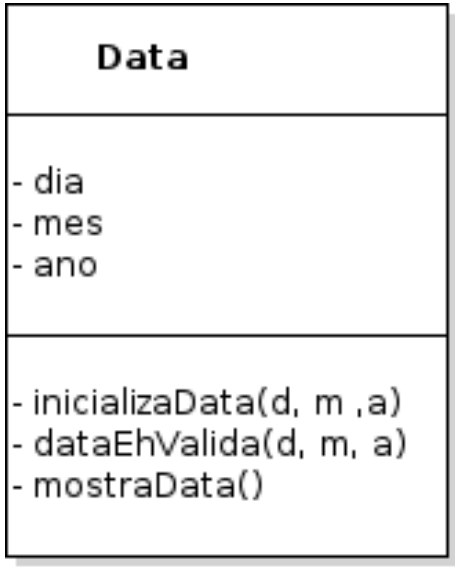
Conta bancária simplificada


```
1  modelo ContaBancariaSimplificada
2  inicio do modelo
3      dado nomeDoCorrentista, saldo, contaEhEspecial;
4
5      // Inicializa todos os dados do modelo
6      operacao abreConta(nome, deposito, ehEspecial)
7          inicio
8              nomeDoCorrentista = nome;
9              saldo = deposito;
10             contaEhEspecial = ehEspecial;
11         fim
12
13     operacao abreContaSimples(nome)
14         inicio
15             nomeDoCorrentista = nome;
16             saldo = 0.0;
17             contaEhEspecial = falso;
18         fim
19
20     // Deposita um valor na conta
21     operacao deposita(valor)
22         inicio
23             saldo = saldo + valor;
24         fim
```

Conta bancária simplificada (Cont.)


```
25      // Retira um valor da conta
26      operacao retira(valor)
27          inicio
28              se(contaEhEspecial == falso) // nao eh especial
29                  inicio
30                      se(valor <= saldo) // se existe saldo
31                          saldo = saldo - valor;
32                  fim
33              senao // Conta especial, pode retirar a vontade
34                  saldo = saldo - valor
35          fim
36
37      // Mostra os dados da conta, imprimindo seus valores
38      operacao mostraDados()
39      inicio
40          imprime "O nome do correntista eh: ";
41          imprime nomeDoCorrentista;
42          imprime "O saldo eh: ";
43          imprime saldo;
44          se(contaEhEspecial) imprime "A conta eh especial";
45          senao imprime "A conta eh comum";
46      fim
47  fim do modelo
```

Modelo Data



Uma variante do diagrama UML para o modelo Data

Modelo: Data

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  modelo Data
2  inicio do modelo
3      dado dia, mes, ano; // componentes da data
4
5      // Inicializa simultaneamente todos os dados do modelo
6      operacao inicializaData(d, m, a)
7          inicio
8              se(dataEhValida(d,m,a))
9                  inicio
10                     dia = d;
11                     mes = m;
12                     ano = a;
13                 fim
14             senao
15                 inicio
16                     dia = 0;
17                     mes = 0;
18                     ano = 0;
19                 fim
20         fim
```

Modelo: Data (Cont.)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
21      // Verifica se a data eh valida
22      operacao dataEhValida(d, m, a)
23      inicio
24          se((dia>=1) e (dia<=31) e (mes>=1) e (mes<=12))
25              retorna verdadeiro;
26          senao
27              retorna falso;
28      fim
29
30      // Mostra a data imprimindo valores de seus dados
31      operacao mostraData()
32          inicio
33              imprime dia;
34              imprime "/";
35              imprime mes;
36              imprime "/";
37              imprime ano;
38          fim
39  fim do modelo
```

Classes e Objetos

Classes

Classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de **funções-membro** e estados possíveis destes objetos através de **atributos**. Resumidamente, classe é o modelo a partir do qual os objetos são feitos.

Classes

Classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de **funções-membro** e estados possíveis destes objetos através de **atributos**. Resumidamente, classe é o modelo a partir do qual os objetos são feitos.

- Analogia: Pense em uma classe como a planta de uma casa. Os objetos são as próprias casas.

Classes

Classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de **funções-membro** e estados possíveis destes objetos através de **atributos**. Resumidamente, classe é o modelo a partir do qual os objetos são feitos.

- Analogia: Pense em uma classe como a planta de uma casa. Os objetos são as próprias casas.

- Para representação de dados específicos usando classes é preciso criar **objetos** ou **instâncias** da classe.
- Um objeto ou instância é uma materialização da classe e, assim, pode ser usado para representar dados e executar operações.

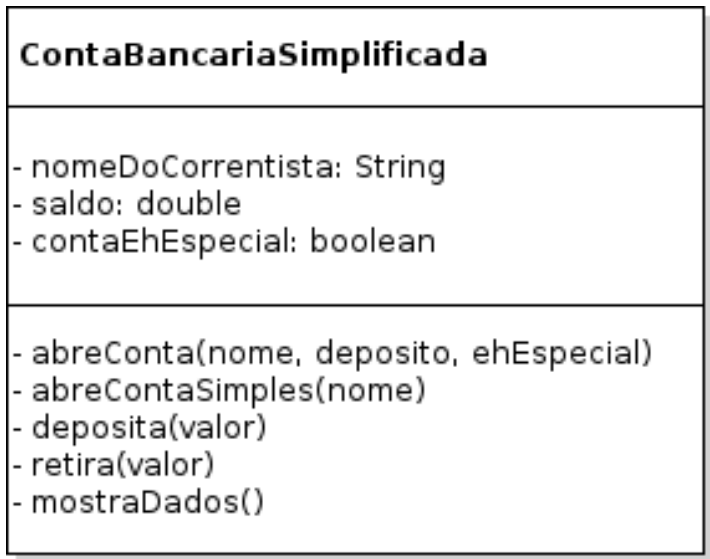
Classes e atributos

- Considerando os exemplos de modelos mostrados anteriormente, os modelos seriam as classes e a partir destas classes poderíamos criar instâncias.

Classes e atributos

- Considerando os exemplos de modelos mostrados anteriormente, os modelos seriam as classes e a partir destas classes poderíamos criar instâncias.

- Os dados contidos em uma classe são chamados de campos ou atributos daquela classe.
  - Cada campo deve ter um nome e ser de um tipo, que será um tipo nativo do Java ou uma classe existente na linguagem ou definida pelo programador.



Classes e funções-membro

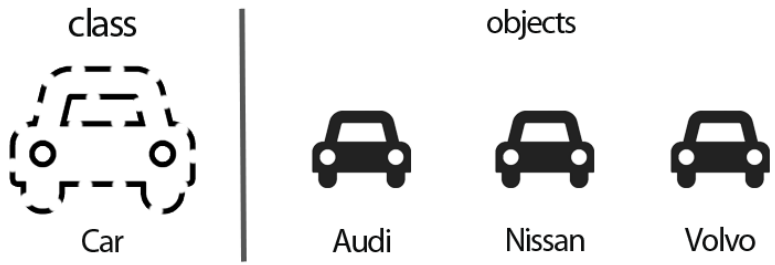
- As operações contidas em uma classe são chamadas de funções-membro dessa classe.
- Funções-membro são geralmente chamados ou executados explicitamente a partir de outros trechos de código na classe que os contém ou a partir de outras classes.

Classes e funções-membro

- As operações contidas em uma classe são chamadas de funções-membro dessa classe.
- Funções-membro são geralmente chamados ou executados explicitamente a partir de outros trechos de código na classe que os contém ou a partir de outras classes.
- Funções-membro podem receber argumentos na forma de valores de tipos nativos de dados ou referências a instâncias de classes. Vários argumentos de vários tipos podem ser fornecidos simultaneamente para uma função.
- Funções-membro podem também retornar valores ou instâncias de classes. Podem não retornar nenhum valor ou retornar um único valor, mas não podem retornar simultaneamente mais do que um valor.

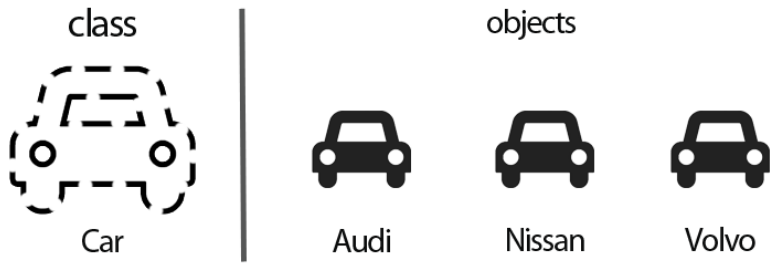
Objetos

- Um objeto ou instância é uma materialização da classe e, assim, pode ser usado para representar dados e executar operações.



Objetos

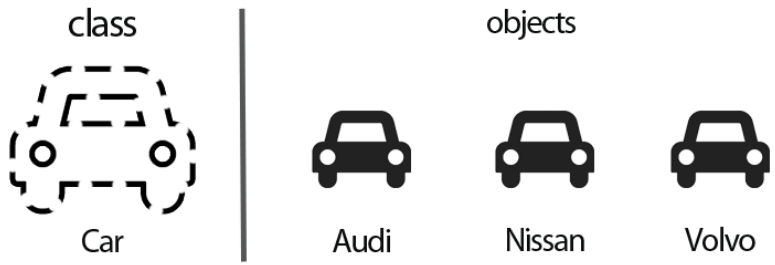
- Um objeto ou instância é uma materialização da classe e, assim, pode ser usado para representar dados e executar operações.



- As classes servem apenas como um modelo. Para que possamos trabalhar com elas, é preciso instanciar objetos a partir da classe. Uma classe um novo tipo de dados.

Objetos

- Um objeto ou instância é uma materialização da classe e, assim, pode ser usado para representar dados e executar operações.



- As classes servem apenas como um modelo. Para que possamos trabalhar com elas, é preciso instanciar objetos a partir da classe. Uma classe um novo tipo de dados.

- Em C++, os objetos podem ser instanciados como variáveis comuns (criação e destruição manipulada pelo compilador) ou podem ser instanciados dinamicamente (criação e destruição manipulada pelo PROGRAMADOR)

Classes em C++

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)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
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)

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
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


```cpp
#include <iostream> // destrutor.cpp

class Ponto {
private:
    double x;
    double y;

public:
    Ponto(double X, double Y) {
        x = X;
        y = Y;
        std::cout << "Ponto construido" << std::endl;
    }

    // Destrutor (note o til antes do nome da funcao)
    ~Ponto() {
        std::cout << "Ponto destruido" << std::endl;
    }

    double getX() { return x; }
    double getY() { return y; }
    void setX(double x) { this->x = x; }
    void setY(double y) { this->y = y; }
};
```

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

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```cpp
#include <iostream> // main2.cpp
#include "Ponto2.h"
using namespace std;

int main() {
    Ponto p1 { 2.3, 4.5 };
    Ponto p2 { 4, 7.8 };
    Ponto p3 = p2;

    cout << "Ponto 1: ";
    cout << "(" << p1.getX() << "," << p1.getY() << ")\n";

    cout << "Ponto 2: ";
    cout << "(" << p2.getX() << "," << p2.getY() << ")\n";

    cout << "Ponto 3: ";
    cout << "(" << p3.getX() << "," << p3.getY() << ")\n";

    cout << "Distancia: " << p1.distancia(p2) << endl;
    return 0;
}
```

Outra Implementação do TAD Ponto

Arquivo Ponto3.h


```
1  #ifndef PONTO3_H
2  #define PONTO3_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

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
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

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM