Classes e Objetos

Roland Teodorowitsch

Programação Orientada a Objetos - ECo - Curso de Engenharia de Computação - PUCRS

8 de junho de 2022

Motivação

Um Sistema para Gestão Escolar

Dados (Atributos)	Funções (Métodos)				
	mediaIdade()	mediaAprovados()	imprimeAlunos()	calculaSalario()	imprimeProfs()
String nomeAlunos[10]					
int idadeAlunos[10]					
float notas[10][3]					
float HORA_AULA=35					
string nomeProfs[5]					
float salarioProfs[5]					
int nTurmas[5]					

Um Sistema para Gestão Escolar

Dados (Atributos)	Funções (Métodos)				
	mediaIdade()	mediaAprovados()	imprimeAlunos()	calculaSalario()	imprimeProfs()
String nomeAlunos[10]			X		
int idadeAlunos[10]	X		X		
float notas[10][3]		X	X		
float HORA_AULA=35				X	
string nomeProfs[5]					X
float salarioProfs[5]				X	X
int nTurmas[5]				X	

Um Sistema para Gestão Escolar

Dados (Atributos)	Funções (Métodos)				
	mediaIdade()	mediaAprovados()	imprimeAlunos()	calculaSalario()	imprimeProfs()
String nomeAlunos[10]			X		
int idadeAlunos[10]	X		X		
float notas[10][3]		X	X		
float HORA_AULA=35				X	
string nomeProfs[5]					X
float salarioProfs[5]				X	X
int nTurmas[5]				X	

Objetos e Classes

Objeto

- Um objeto refere-se a uma coisa do mundo
- Agrupa dados e funções correlatas
- Constituído por atributos e métodos
- Atributos são os dados (variáveis de instância)
- Métodos são funções e procedimentos
  - Operam sobre atributos
  - Definem o comportamento do objeto

Exemplos de Objetos

- Em C++, string é uma classe

```
string str = "Strings em C++ sao objetos";
cout << str.length() << endl;
```

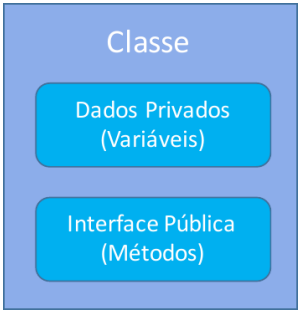
- Objetos criados a partir de uma classe chamada Aluno

```
Aluno a1;
a1.defineNome("Joao Souza");

Aluno *a2 = new Aluno("Paula da Silva");
cout << a2->obtemNome() << endl;
delete a2;
```

Classe

- Uma classe descreve um conjunto de objetos com mesmo comportamento
- Objetos não são programados individualmente
- O programador escreve classes
- Objetos são instâncias de suas classes
- Uma classe descreve atributos e métodos de um conjunto de objetos



Resumo

- **Objetos** representam coisas do mundo
  - Têm atributos e métodos
- **Classes** especificam objetos
  - Definem atributos e métodos

Resumo

- **Objetos** têm um estado interno e ...
  - Valores atuais de seus atributos
- ... uma interface pública.
  - Conjunto de métodos para manipulação do estado interno
- Interface pública encapsula estado interno
  - Esconde detalhes de implementação
- **Classe** define como são estruturados o estado interno e a interface pública (e sua implementação)

Implementando uma Classe Simples

Classe Contador

- Uma classe que modela um dispositivo mecânico que é usado para realizar contagens
  - Por exemplo, para contar quantas pessoas estão assistindo a um concerto ou quantas pessoas embarcaram em um ônibus
- O que deve ser feito?
  - Incrementar o dispositivo
  - Obter o valor atual
  - Zerar o contador

Criando a Classe Contador

- Estrutura

```
class Contador {
  // ...
};
```

- Atributos
- Métodos
  - De Acesso (*getters*)
  - De Modificação (*setters*)
  - Construtor(es)
  - Destrutor

Atributos

- Por enquanto, serão sempre privados
- Cada objeto da classe `Contador` terá a sua cópia dos atributos

```
class Contador {
  private:
    int valor;
  // ...
};
```

Métodos de Acesso (*Getters*)

- Apenas acessam atributos do objeto, sem alterar o estado interno do objeto
- Usualmente retornam UM atributo
- Comumente nomeados como "get" + AttributeName ou "obtem" + NomeAtributo
- Raramente apresentam parâmetros e costumam ser públicos

```
class Contador {
  private:
    int valor;
  public:
    // ...

    int obtemValor() {
      return valor;
    }

    // ...
};
```

Métodos de Modificação (*Setters*)

- Modificam atributos do objeto, alterando o estado interno do objeto
- Geralmente apresentam um parâmetro e alteram UM atributo do objeto
- Normalmente nomeados como "set" + AttributeName ou "define" + NomeAtributo
- Frequentemente o tipo de retorno é `void` e também costumam ser públicos

```
class Contador {
  private:
    int valor;
  public:
    // ...

    void incrementa() {
      valor = valor + 1;
    }

    void defineValor(int v) {
      valor = v;
    }

    // ...
};
```

Atributos e Métodos

- Observe que:
  - `private:` e `public:` são modificadores
  - Por enquanto, vamos usar `private:` para atributos e `public:` para métodos
  - Mas NÃO é raro encontrar métodos `private:` (são de uso interno)
- Objetos sem *setter* são imutáveis
- É comum cada atributo ter pelo menos um método *getter* e um método *setter*, mas...
- Deve-se evitar criar esses métodos mecanicamente sem verificar se fazem sentido ou NÃO
- Deve-se sempre privilegiar métodos de negócio
  - Em vez de `conta.defineSaldo(conta.obtemSaldo() + 100)`
  - É melhor `conta.credita(100)`
- Dica de leitura:
  `https://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/`

Método Construtor

- É um método que será chamado quando um objeto da classe for criado (por exemplo, quando se usa o operador `new`)
- Constrói objetos, definindo os valores iniciais dos atributos (ou seja, inicializando o estado interno do objeto)
- Tem o mesmo nome da classe
- NÃO tem tipo de retorno
- Geralmente é público
- Uma classe pode ter mais de um construtor (sobrecarga)
- Se nenhum construtor for declarado, o compilador criará um construtor padrão automaticamente (vazio)
  - Ele NÃO receberá nenhum parâmetro
  - Ele NÃO inicializará as variáveis de instância

Método Construtor

```
class Contador {
  private:
    int valor;

  public:
    Contador() {
      valor = 0;
    }

    Contador(int v) {
      valor = v;
    }

    int obtemValor() {
      return valor;
    }

    void incrementa() {
      valor = valor + 1;
    }

    void defineValor(int v) {
      valor = v;
    }
};
```

Construtor com Parâmetros Opcionais

- É possível definir métodos e construtores com valores opcionais (*default*)

```
class Contador {
  private:
    int valor;

  public:
    Contador(int v = 0) {
      valor = v;
    }

    // ...
};
```

Execução de Construtores

- Objetos podem ser criados como variáveis ou usando o operador `new`

```
int main() {
  Contador c1; // Novo contador com valor inicial 0
  Contador *c2 = new Contador(10); // Contador com valor 10
  c2->incrementa();
  c2->incrementa();
  cout << c2->obtemValor() << endl; // Deve imprimir 12
  delete c2;
  c1.incrementa();
  c1.incrementa();
  c1.incrementa();
  cout << c1.obtemValor() << endl; // Deve imprimir 3
  return 0;
}
```

Construtores – Erros Comuns

- NÃO inicializar referências a objetos
  - **Referências** são inicializadas por padrão com `null`
  - Chamar um método de uma referência que contém `null` resulta em um erro de execução (*segmentation fault*)

```
int main() {
  Contador *c2;
  c2->incrementa(); // SEGMENTATION FAULT
  cout << c2->obtemValor() << endl;
  delete c2;
  return 0;
}
```

Construtores – Erros Comuns

- Tentar chamar um construtor
  - NÃO se pode chamar um construtor como é feito com outros métodos
  - Ele é "invocado" automaticamente na declaração de um objeto ou através do operador `new`

```
Contador c1, c2(10);
Contador *c3 = new Contador();
```

  - NÃO se pode invocar um construtor para um objeto que já existe

```
c1.Contador(); // ERRO
c3->Contador(); // ERRO
```

  - Mas pode-se criar um novo objeto usando uma referência existente

```
Contador *c3 = new Contador();
c3->incrementa();
delete c3;
c3 = new Contador(10);
```

Construtores – Erros Comuns

- Declarar um construtor como `void`
  - Construtores NÃO têm tipo de retorno

```
class Contador {
  private:
    int valor;
  public:
    void Contador() {
      valor = 0;
    }
    // ...
};
```

Destrutor

- É um método usado para "limpeza" (liberação de recursos) quando um objeto não é mais necessário
- NÃO tem parâmetros
- Também NÃO tem tipo de retorno
- Seu nome corresponde ao nome da classe precedido de um til (~)
- É invocado quando um objeto é destruído ou quando uma referência é desalocada (delete)

```
class Contador {
  // ...
  public:
    ~Contador() {
      cout << "Destrutor de Contador chamado..." << endl;
    }
    // ...
};
```

Organizando o Código das Classes

- A declaração de uma classe
  - Contém **atributos privados** e **métodos públicos**
  - Deve ser terminada por ponto-e-vírgula (;)

```
class NomeDaClasse {
  private:
    // atributos
  public:
    // metodos
};
```

- É possível incluir a declaração de várias classes em um mesmo arquivo
- Também é possível separar **definição** (que fica em um arquivo .hpp) de **implementação** (que fica em um arquivo .cpp)

Classe com Arquivo de Cabeçalho

```
// Contador.hpp
#ifndef _CONTADOR_HPP
#define _CONTADOR_HPP
using namespace std;
class Contador {
  private:
    int valor;
  public:
    Contador(int v = 0);
    ~Contador();
    int obtemValor();
    void incrementa();
    void defineValor(int v);
};
#endif
```

```
// main.cpp
#include "Contador.hpp"
#include <iostream>
int main() {
  Contador *c1 = new Contador(10);
  cout << c1->obtemValor() << endl;
  delete c1;
  return 0;
}
```

```
// Contador.cpp
#include <iostream>
#include "Contador.hpp"

Contador::Contador(int v) { // Observar o uso de ::
  valor = v;
}
Contador::~Contador() {
  cout << "Destrutor de Contador chamado..." << endl;
}
int Contador::obtemValor() {
  return valor;
}
void Contador::incrementa() {
  valor = valor + 1;
}
void Contador::defineValor(int v) {
  valor = v;
}
```

Funções e Métodos Embutidos

- A palavra-chave `inline` colocada no início da declaração ou da implementação de funções ou métodos, faz com que o compilador substitua a chamada pela própria implementação
- Isto torna o programa mais rápido, pois evita a execução da chamada
- Um método definido no corpo de uma declaração de classe é implicitamente embutido

```
class Exemplo {
private:
    int valor;
public:
    Exemplo(int v) {
      valor = v;
      cout << "Construtor (implicitamente embutido) chamado..." << endl;
    }
    ~Exemplo();
    int obtemValorNaoEmbutido();
    int obtemValorEmbutido();
};

Exemplo::~Exemplo() {  cout << "Destrutor (NAO embutido) chamado..." << endl;  }

int Exemplo::obtemValorNaoEmbutido() {  return valor;  }

inline int Exemplo::obtemValorEmbutido() {  return valor;  }
```

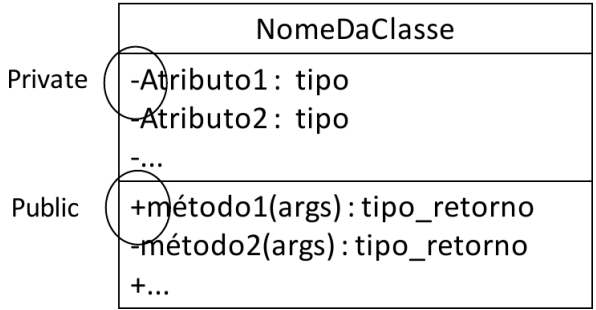
Introdução à UML

Unified Modeling Language (UML)

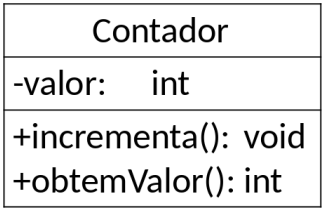
- Linguagem para modelagem de sistemas
- Visual e abstrata
- Diversos tipos de diagramas
- Nesta disciplina usaremos Diagrama de classes

Diagrama de Classes

- Representação visual de classes
- Classes representadas como caixas
- Atributos e métodos (sem implementação)



Exemplo – Classe Contador




```
class Contador {
  private:
    int valor;

  public:
    void incrementa() {
      valor = valor + 1;
    }
    int obtemValor() {
      return valor;
    }
};
```

Construindo uma Classe

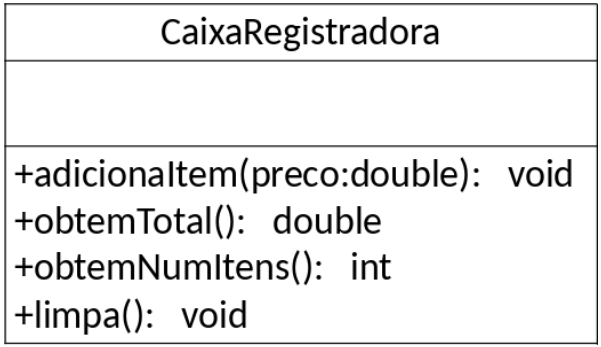
Passos Gerais para Construir uma Classe

1. Definir a interface pública (métodos)
2. Definir os atributos (dados)
3. Implementar a interface pública

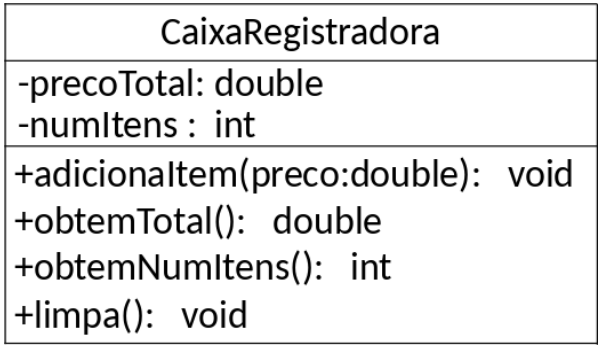
Exemplo – Caixa Registradora

- Comportamento desejado:
  - Adicionar o preço de um item
  - Pegar o total da venda
  - Pegar o total de itens vendidos
  - Limpar o caixa para uma nova venda

(1) Definindo a Interface Pública (Métodos)



(2) Definindo Atributos (Dados)



(3) Implementação em 3 arquivos

1. CaixaRegistradora.hpp
2. CaixaRegistradora.cpp
3. main.cpp

```
#include <iostream>
#include "CaixaRegistradora.hpp"
using namespace std;
int main () {
  CaixaRegistradora * caixa = new CaixaRegistradora ();
  caixa->adicionaItem (1.99); // Adiciona primeiro item
  caixa->adicionaItem (2.99); // Adiciona segundo item
  caixa->adicionaItem (1.50); // Adiciona terceiro item
  cout << caixa->obtemTotal () << endl; // Saida : 6.48
  cout << caixa->obtemNumItens () << endl; // Saida : 3
  caixa->limpa ();
  cout << caixa->obtemTotal () << endl;; // Saida : 0
  cout << caixa->obtemNumItens () << endl; // Saida : 0
  delete caixa;
  CaixaRegistradora caixa2;
  caixa2.adicionaItem(123.456);
  cout << caixa2.obtemTotal() << endl;
  cout << caixa2.obtemNumItens() << endl;
  return 0;
}
```

Compilar com: `g++ -std=c++11 CaixaRegistradora.cpp main.cpp -o executavel`

Lista de Exercícios 1

Lista de Exercícios 1

1. Implementar os arquivos `CaixaRegistradora.hpp` e `CaixaRegistradora.cpp` tal como apresentado nos diagramas UML e exemplos das lâminas anteriores.
2. Crie uma classe para representar uma pessoa, com os seguintes atributos privados: nome, idade e altura. Crie os métodos públicos necessários para acessar estes atributos, alterar estes atributos e também um método para imprimir os dados de uma pessoa.

Lista de Exercícios 2

Lista de Exercícios 2 – Exercício 1

1. Crie uma classe denominada **Elevador** para armazenar as informações de um elevador dentro de um prédio.
   A classe deve armazenar o andar atual (0=térreo), total de andares no prédio, excluindo o térreo, capacidade do elevador, e quantas pessoas estão presentes nele.
   A classe deve também disponibilizar os seguintes métodos:
   - **inicializa**: que deve receber como parâmetros: a capacidade do elevador e o total de andares no prédio (os elevadores sempre começam no térreo e vazios);
   - **entra**: para acrescentar uma pessoa no elevador (só deve acrescentar se ainda houver espaço);
   - **sai**: para remover uma pessoa do elevador (só deve remover se houver alguém dentro dele);
   - **sobe**: para subir um andar (não deve subir se já estiver no último andar);
   - **desce**: para descer um andar (não deve descer se já estiver no térreo);
   - **obtem...**: métodos para obter cada um dos os dados armazenados.

Lista de Exercícios 2 – Exercício 2

2. Implemente uma classe chamada **Televisor** para simular o funcionamento de um aparelho televisor. O televisor tem um controle de volume do som e um controle de seleção de canal. Considere que:
   - O controle de volume permite aumentar ou diminuir a potência do volume de som em uma unidade de cada vez.
   - O controle de canal também permite aumentar e diminuir o número do canal em uma unidade, porém, também possibilita trocar para um canal indicado.
   - Também devem existir métodos para consultar o valor do volume de som e o canal selecionado.

   No programa principal, crie uma televisão e troque de canal algumas vezes. Aumente um pouco o volume, e exiba o valor de ambos os atributos.

Lista de Exercícios 2 – Exercício 3

3. Crie uma classe em C++ chamada **Relogio** para armazenar um horário, composto por hora, minuto e segundo. A classe deve representar esses componentes de horário e deve apresentar os métodos descritos a seguir:
   - um método chamado **defineHora**, que deve receber o horário desejado por parâmetro (hora, minuto e segundo);
   - um método chamado **obtemHora** para retornar o horário atual, através de 3 variáveis passadas por referência;
   - um método para avançar o horário para o próximo segundo (lembre-se de atualizar o minuto e a hora, quando for o caso).

Lista de Exercícios 3 – Exercício 4

4. Implemente um condicionador de ar. O condicionador possui 10 potências diferentes. Cada unidade da potência do condicionador diminui a temperatura do ambiente em 1.8°C. A variação que o condicionador consegue causar está no intervalo [0°C - 18°C], ou seja, zero graus de variação quando desligado e dezoito graus de variação quando ligado na potência máxima.
   Através de um sensor, o condicionador é informado da temperatura externa. Dada essa temperatura e a potência selecionada, o condicionador calcula e retorna a temperatura do ambiente.
   No programa principal, crie dois condicionadores. Informe duas temperaturas externas diferentes para cada um (Exemplo: 25°C e 31°C), ajuste o segundo em potência máxima (10) e o primeiro em potência média (5). Finalmente, exiba a temperatura resultante de cada ambiente.

Lista de Exercícios 2 – Exercício 5

5. Implemente um carro. O tanque de combustível do carro armazena no máximo 50 litros de gasolina. O carro consome 15 km/litro. Deve ser possível:
   - Abastecer o carro com uma certa quantidade de gasolina;
   - Mover o carro em uma determinada distância (medida em km);
   - Retornar a quantidade de combustível e a distância total percorrida.

   Deve-se criar um método **toString** para armazenar os dados da classe em um string.
   No programa principal, crie 2 carros. Abasteça 20 litros no primeiro e 30 litros no segundo. Desloque o primeiro em 200 km e o segundo em 400 km. Exiba na tela a distância percorrida e o total de combustível restante para cada um (utilizar o método criado **toString**).
   Dica: Para facilitar a implementação do método **toString**, utilize o tipo **stringstream** da biblioteca `sstream`.

Créditos

Créditos

- Estas lâminas contêm trechos de materiais disponibilizados pelos professores Rafael Garibotti, Daniel Callegari, Sandro Fiorini e Bernardo Copstein.
- Os exercícios das Listas de Exercícios 2 e 3 foram elaborados pelo professor Márcio Sarroglia Pinho.