

Curso de Extensão - INF1900

Programação em C++



Módulo 2

Programação Orientada a Objetos em C++ Aula 1

Profa. Dra. Esther Luna Colombini

esther@ic.unicamp.br

Agosto de 2023

Módulo 2: Programação Orientada a Objetos em C++ (10h)



Profa. Dra. Esther Luna Colombini



Introduzir os conceitos fundamentais da programação orientada a objetos em C++ e capacitar os alunos a projetar e implementar programas orientados a objetos

- Princípios da programação orientada a objetos (POO)
- Classes e objetos em C++
- Encapsulamento, herança e polimorfismo
- Construtores e destrutores
- Sobrecarga
- Relacionamentos

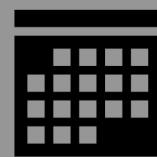
Monitorias

- **Monitores do Módulo:**
 - Alana Correia
 - Iury Cleveston
- **Dia/Horário do atendimento dos monitores:**
 - A definir em conjunto
- **Dia/Horário do atendimento do professor:**
 - Quintas às 18:00h

Calendário

Aulas: segunda-feira e quarta-feira

Horário: 8:00h às 10:00h



28/08/23	MÓDULO 2
30/08/23	MÓDULO 2
31/08/23	MÓDULO 2 - ATENDIMENTO
04/09/23	MÓDULO 2
06/09/23	MÓDULO 2
11/09/23	MÓDULO 2 - ATENDIMENTO

Avaliação

- **Avaliação:**
 - Atividades práticas a serem realizadas em dupla liberadas todas às quartas
- **Média final:** média aritmética das notas

Bibliografia

- Stroustrup, B. (2013). The C++ Programming Language (4th Edition). Addison-Wesley Professional.
- Josuttis, N. (2019). C++17 - The Complete Guide: First Edition. Leanpub.
- Schildt, H. (2017). C++: The Complete Reference (5th Edition). McGraw-Hill Education.

O que é POO?

Programação Orientada a Objetos

- Objetivos:
 - Entender
 - o que é POO
 - para que serve
 - porque OO

Programação Orientada a Objetos

- Paradigmas de Programação
- Histórico
- Orientação a Objetos

Paradigmas de Programação

- Orientado a procedimento (Procedural)
 - procedimentos, sequência
- Orientado a lógica
 - regras, padrões e inferências
- Funcional
 - funções matemáticas
- Orientado a objeto
 - abstração, ligação dinâmica, herança

Orientado a procedimento

- Ênfase maior dada aos procedimentos e funções
 - Modela-se a solução de problemas com base nas funções a serem executadas
 - Dados tratados de forma secundária
 - Paradoxo
 - Dados são mais importantes
 - Sem os dados os procedimentos não teriam utilidade prática
 - Exemplos
 - Pascal, ALGOL, C

Orientado a procedimento

```
typedef unsigned long NumConta;
typedef int bool;

bool fazDeposito(NumConta conta, float valor);
float fazRetirada(NumConta conta, float valor);
bool transfere(NumConta origem, NumConta destino, float valor);
void imprimeConta(NumConta conta);

struct Conta
{
    char *titular;
    NumConta contaId;
    float saldo;
    char tipo;
};|
```

Paradigma Lógico

- Programação Lógica
 - Programação de forma declarative, ou seja, especificando o que deve ser computado ao invés de como deve ser computado
 - Relações são mais genéricas do que mapeamentos, portanto programação lógica é mais alto nível que imperativa ou funcional
 - Sem instruções explícitas e sequenciamento
 - Exemplos
 - PROLOG

Paradigma Lógico

```
predecessor(X,Z) :- parent(X,Z).  
predecessor(X,Z) :-  
    parent(X,Y), predecessor(Y,Z).  
sister(X,Y) :- female(X), parent(Z,X), parent(Z,Y), not(X=Y).  
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).|
```

Paradigma Funcional

- Uso de expressões e funções no lugar de variáveis, comandos e procedimentos
 - enfatiza a avaliação de expressões
 - não utiliza comandos e algoritmos
 - não utiliza variáveis e atribuições
 - exige bastante disciplina de programação
 - produz programas que podem ser mais facilmente verificados
 - Exemplos
 - Lisp, ML, Haskell

Paradigma Funcional

```
import re
from collections import Counter
from functools import reduce

# Função para normalizar o texto (remover pontuações e transformar em minúsculas)
def normalizar_texto(texto):
    return re.findall(r'\w+', texto.lower())

# Função para contar a frequência das palavras
def contar_palavras(texto):
    palavras = normalizar_texto(texto)
    contador = Counter(palavras)
    return contador

# Função para filtrar palavras com base na frequência mínima
def filtrar_por_frequencia(contador, frequencia_minima):
    return dict(filter(lambda item: item[1] >= frequencia_minima, contador.items()))

# Função para juntar contadores
def juntar_contadores(contador1, contador2):
    return contador1 + contador2
```

Paradigma Funcional

```
# Texto de exemplo
texto = """
O paradigma funcional é uma abordagem de programação que trata a computação como avaliação
de funções matemáticas e evita a mudança de estado e dados mutáveis. Em contraste, o
paradigma imperativo modifica o estado do programa através de atribuições.
"""

# Contagem de palavras individuais no texto
contador_palavras = contar_palavras(texto)

# Contagem de palavras filtradas por frequência mínima
frequencia_minima = 2
palavras_filtradas = filtrar_por_frequencia(contador_palavras, frequencia_minima)

# Imprimir as palavras e suas frequências
print(palavras_filtradas)
```

Orientado a Objetos

- Programação em POO
 - Ênfase à estrutura de dados, adicionando funcionalidade a elas
- Inicia-se decidindo os objetos necessários, que são, então, caracterizados através de propriedades e comportamento
- O programador vê seu programa como uma coleção de objetos cooperantes comunicando-se através de mensagens
- Exemplos
 - C++, Java, C#

Orientado a Objetos

```
class Conta
{
public:
    bool fazDeposito(float valor);
    float fazRetirada(float valor);
    bool transfere(Conta &destino, float valor);
    void imprimeConta() const;
private:
    char titular[255];
    int numeroConta;
    float saldo;
};
```

Paradigmas: Comparativo

```
typedef unsigned long NumConta;
typedef int bool;

bool fazDeposito(NumConta conta, float valor);
float fazRetirada(NumConta conta, float valor);
bool transfere(NumConta origem, NumConta destino, float valor);
void imprimeConta(NumConta conta);

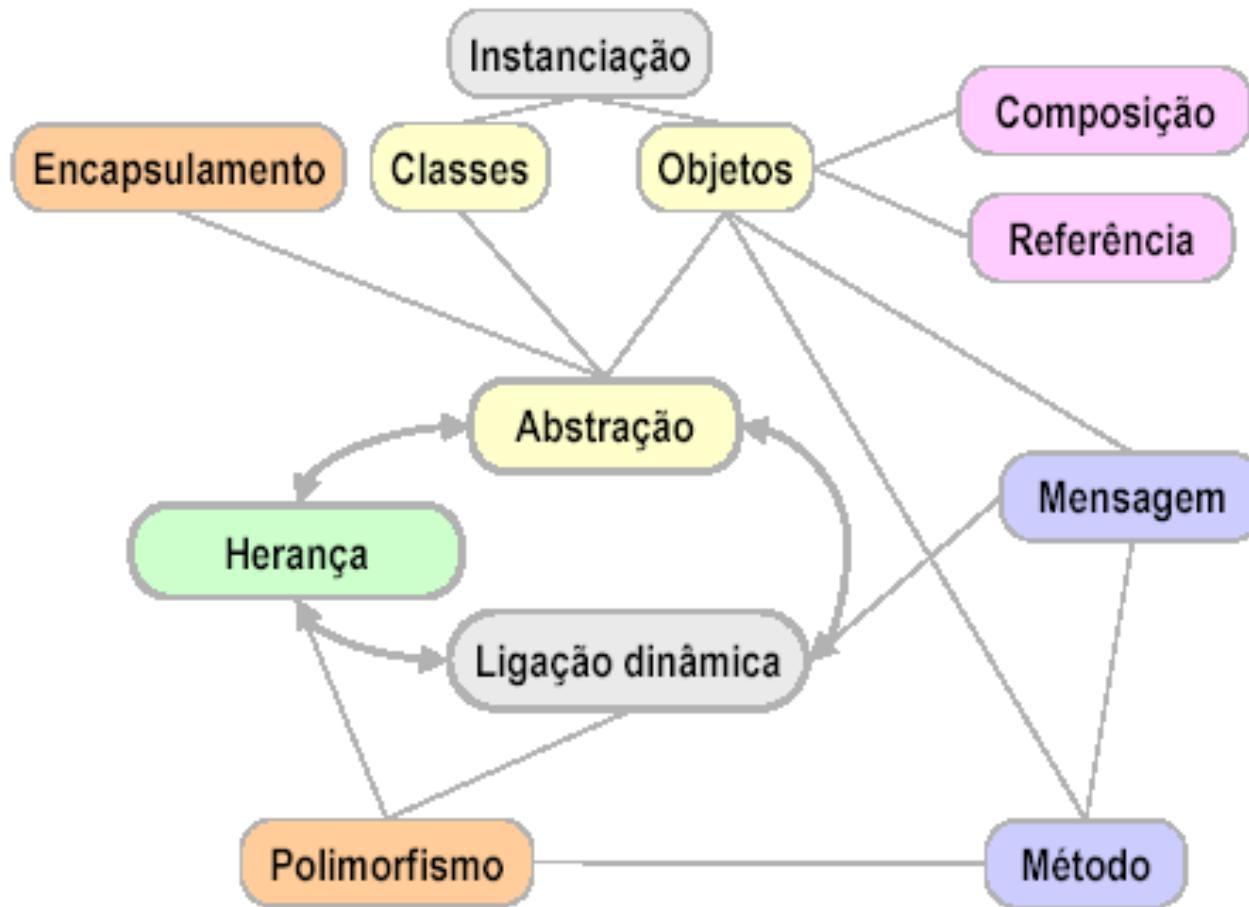
struct Conta
{
    char *titular;
    NumConta contaId;
    float saldo;
    char tipo;
};

};|
```

```
class Conta
{
public:
    bool fazDeposito(float valor);
    float fazRetirada(float valor);
    bool transfere(Conta &destino, float valor);
    void imprimeConta() const;
private:
    char titular[255];
    int numeroConta;
    float saldo;
};

};
```

Orientação a Objetos



Histórico da OO

- Orientação a Objetos não é um conceito novo
- O conceito de objetos e classes foi primeiro utilizado na linguagem Simula 67 (extensão de ALGOL 60)
- Na década de 70, Barbara Liskov trabalhou com TADs e desenvolveu a linguagem CLU
- Durante a década de 70 pesquisadores do Centro de Pesquisa Xerox Palo Alto desenvolveram a linguagem SmallTalk
 - Linguagem mais representativa (pura)

Histórico da OO

- Por volta de 1983, Bjarne Stroustrup incorpora classes a C, criando assim a linguagem C++
 - C++ sofreu várias melhorias nas décadas de 80 e 90. Foi iniciado o processo de padronização em 1991 pela ANSI/ISO
- Na década de 80 surgiram inúmeras metodologias de desenvolvimento
- Em meados da década de 90 (1995) surge a linguagem Java, criada por James Gosling
- Em 1997 Booch, Rumbaugh e Jacobson unificam suas metodologias e criam a UML
- Em 1999, a convite da Microsoft, Anders Hejlsberg formou uma equipe de programadores para desenvolver uma nova linguagem de programação
- Mais tarde, em 2003, tornou-se padrão também da ISO, recebendo a especificação de ISO/IEC 23270.

Histórico de POO



SIMULA



Ole-Johan Dahl (1970)



Kristen Nygaard (1970)

6 /
84

C++



Bjarne Stroustrup

Smalltalk



Alan Kay

JAVA



James Gosling

95

Orientação a Objetos

- Objetos
 - Seres humanos classificam o mundo em objetos



Identidade: ferrari	
Estado	Comportamento
Marca	Acelerar
Cor	Ligar
Motor	Verificar Combustível
Chassi	Calibrar Pneus

Orientação a Objetos



Identidade: cachorro

Estado	Comportamento
Nome	Latir
Raça	Comer
Cor	Brincar
Pedigree	Deitar
Altura	Rolar
Peso	

Orientação a Objetos



Identidade: bicicleta

Estado	Comportamento
Marca	Trocar Marcha
Modelo	Pedalar
Marchas	Freiar
Cor	

Orientação a Objetos

Objetos

Entidade autônoma que une a representação da informação (estruturas de dados) com a sua manipulação (procedimentos)

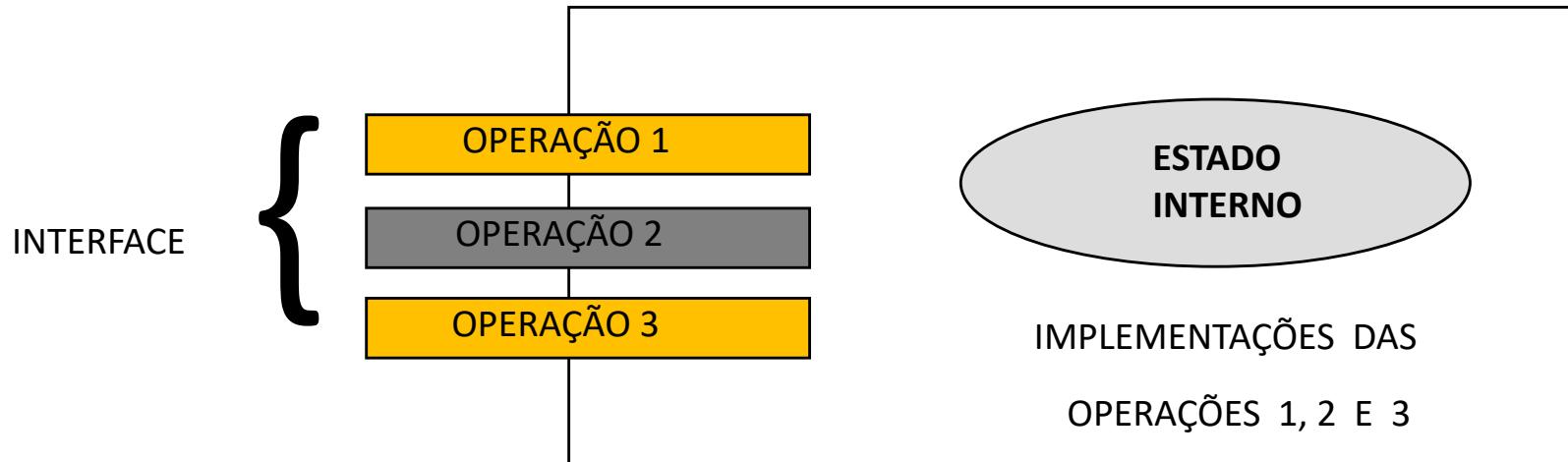
Informalmente um objeto representa uma entidade física, conceitual ou de software

Possui capacidade de processamento e armazena um estado local

Conceito mais próximo na programação convencional

- variável

Estrutura de um Objeto

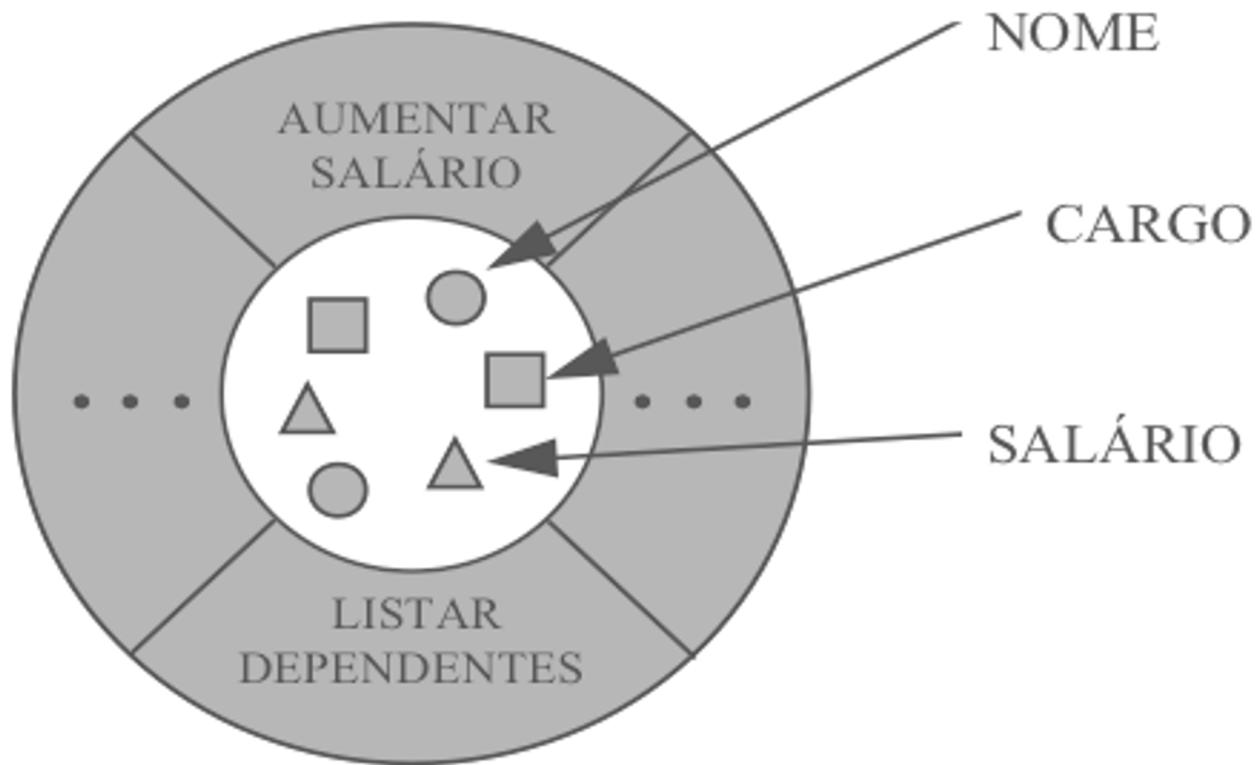


- Em OO os conceitos de dados e procedimentos são reunidos em uma única entidade: o objeto

Estrutura de um Objeto

- Um objeto é composto de:
 - **Propriedades:** informações (estruturas de dados) que representam o estado interno
 - **Comportamento:** conjunto de operações (métodos), que agem sobre as suas propriedades
 - **Identidade:** é uma propriedade que permite diferenciar um objeto de outro, independentemente de sua classe ou estado atual

Estrutura de um Objeto



Benefícios da OO

- Abstração
- Modularidade
- Encapsulamento
- Reusabilidade
- Escalabilidade

Abstração



- Técnica para lidar com a complexidade de um problema
- Destaca os aspectos importantes do objeto real abstraído segundo o observador
- Ignora detalhes não relevantes para o observador
 - Depende da perspectiva do observador

Exemplos de Abstração

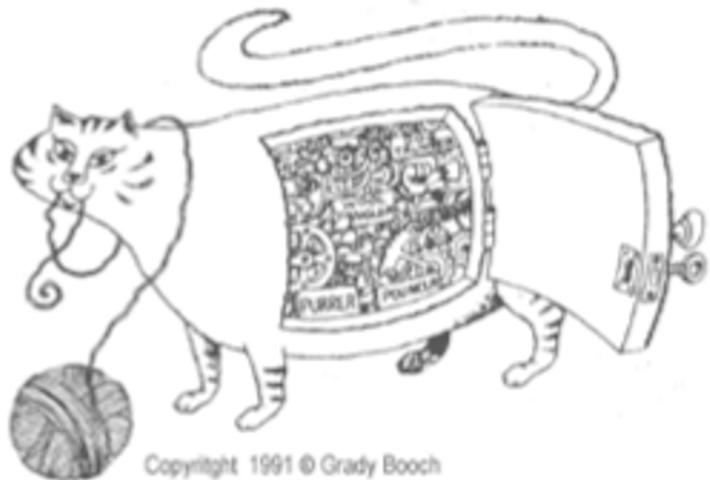
- Um mapa mundi
- Um modelo de avião
- Uma maquete de edifício
- Um tipo de dado abstrato
- A planta baixa de uma casa

Modularidade



- Construção de programas a partir da junção de partes menores (módulos), independentes
- É a propriedade que um sistema tem de poder ser decomposto em um conjunto de módulos coesos e fortemente ligados, facilitando sua compreensão.
- Divide um sistema complexo em módulos menores e melhor gerenciáveis individualmente.

Encapsulamento



Copyright 1991 © Grady Booch

- Abstração e encapsulamento são conceitos complementares: a abstração representa comportamento observável do objeto e o encapsulamento, a implementação deste comportamento.
- O encapsulamento é também chamado de ocultamento da informação
 - os segredos de um objeto que não contribuem para a definição de suas características essenciais (seus dados e operações) são escondidos, ou seja, cliente só conhece a interface.
- A interação de um objeto com o meio externo é realizada exclusivamente através de seus métodos.

Reusabilidade

- Reusabilidade de código entre os objetos da classe
 - Todos os objetos instanciados a partir de uma classe incorporam as suas propriedades e seu comportamento
 - Caso não existissem classes, para cada novo objeto criado, seria preciso uma definição completa do objeto.
- Reusabilidade de código de terceiros

Classes

- Uma classe
 - é um padrão para uma categoria de itens estruturalmente idênticos e um mecanismo para criar estes itens, as instâncias, baseando-se neste padrão.
- **Objetos de estrutura e comportamento** idênticos são descritos em classes
- A descrição das propriedades pode ser feita de uma só vez, independente da quantidade de objetos a serem criados
 - Cada objeto criado a partir de uma classe é denominado de instância
 - Cada instância pertence a uma classe e uma classe pode possuir múltiplas instâncias

Classes

- A Classe é um **modelo** para os objetos
 - Na criação de um objeto ele recebe cópia das estruturas de dados, mas os métodos residem nas classes
 - Métodos definem o comportamento dos objetos da classe e este é único
 - Economiza-se o espaço que seria ocupado pelo código dos métodos em cada objeto da classe

Classes

DATA DE MATRÍCULA
Consultar o sítio da Pós-Graduação

MINISTÉRIO DA DEFESA
INSTITUTO TECNOLÓGICO DE AERONÁUTICA
PRO-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA

FORMULÁRIO DE INSCRIÇÃO DE DISCIPLINA ISOLADA

A. DADOS PESSOAIS:

NOME COMPLETO: _____
Estado civil: _____ Sexo: Masculino Feminino
Data de nascimento: _____ / _____ / _____ Cidade: _____ UF: _____
FILIAÇÃO: Pai: _____ Mão: _____
ENDERÉSCO RESIDENCIAL: _____ Telefone: _____ E-mail: _____
ENDERÉSCO COMERCIAL: _____ Telefone: _____ E-mail: _____
NACIONALIDADE: Brasileiro Estrangeiro Naturalizado
RG: _____ Passaporte N°: _____ CIE: _____
Órgão Emissor: _____ Visto validade: _____ / _____ / _____ Validade: _____ / _____ / _____
CC: _____
Se inscrevo Disciplina Isolada ITA: Sim Não
Em caso afirmativo, liste siglas de disciplinas cursadas no IF Ano: _____

DATA DE MATRÍCULA
Consultar o sítio da Pós-Graduação

MINISTÉRIO DA DEFESA
INSTITUTO TECNOLÓGICO DE AERONÁUTICA
PRO-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA

FORMULÁRIO DE INSCRIÇÃO DE DISCIPLINA ISOLADA

A. DADOS PESSOAIS:

NOME COMPLETO: **DARTH VADER**
Estado civil: _____ Sexo: Masculino Feminino
Data de nascimento: _____ / _____ / _____ Cidade: _____ UF: _____
FILIAÇÃO: Pai: _____ Mão: _____
ENDERÉSCO RESIDENCIAL: _____ Telefone: _____ E-mail: _____
ENDERÉSCO COMERCIAL: _____ Telefone: _____ E-mail: _____
NACIONALIDADE: Brasileiro Estrangeiro Naturalizado
RG: _____ Passaporte N°: _____ CIE: _____
Órgão Emissor: _____ Visto validade: _____ / _____ / _____ Validade: _____ / _____ / _____
CC: _____
Se inscrevo Disciplina Isolada ITA: Sim Não
Em caso afirmativo, liste siglas de disciplinas cursadas no IF Ano: _____

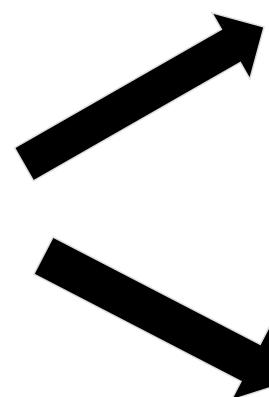
DATA DE MATRÍCULA
Consultar o sítio da Pós-Graduação

MINISTÉRIO DA DEFESA
INSTITUTO TECNOLÓGICO DE AERONÁUTICA
PRO-REITORIA DE PÓS-GRADUAÇÃO E PESQUISA

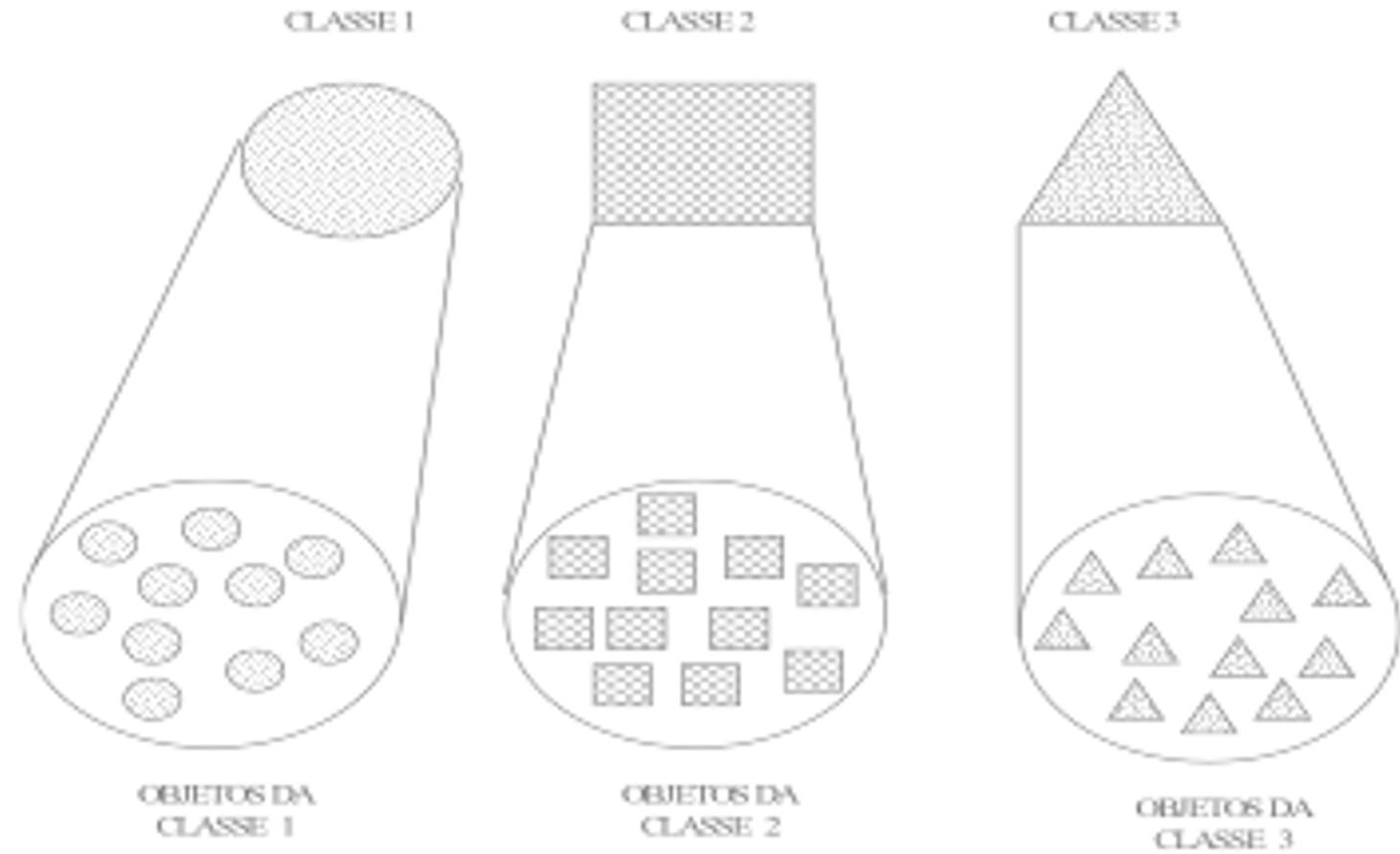
FORMULÁRIO DE INSCRIÇÃO DE DISCIPLINA ISOLADA

A. DADOS PESSOAIS:

NOME COMPLETO: **YODA**
Estado civil: _____ Sexo: Masculino Feminino
Data de nascimento: _____ / _____ / _____ Cidade: _____ UF: _____
FILIAÇÃO: Pai: _____ Mão: _____
ENDERÉSCO RESIDENCIAL: _____ Telefone: _____ E-mail: _____
ENDERÉSCO COMERCIAL: _____ Telefone: _____ E-mail: _____
NACIONALIDADE: Brasileiro Estrangeiro Naturalizado
RG: _____ Passaporte N°: _____ CIE: _____
Órgão Emissor: _____ Visto validade: _____ / _____ / _____ Validade: _____ / _____ / _____
CC: _____
Se inscrevo Disciplina Isolada ITA: Sim Não
Em caso afirmativo, liste siglas de disciplinas cursadas no IF Ano: _____

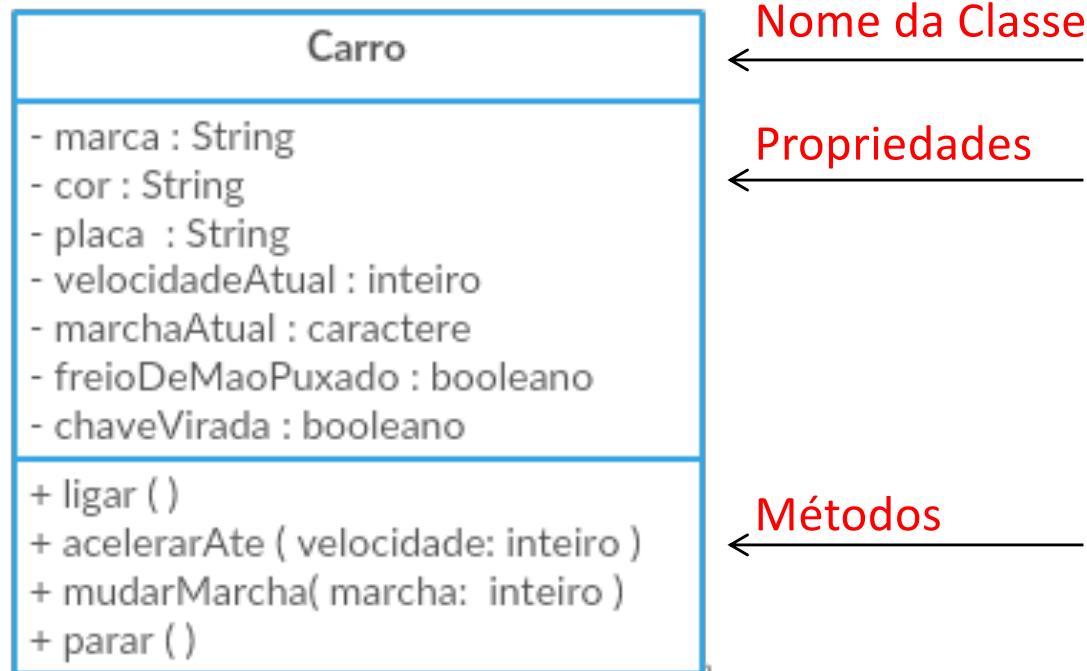


Classes



Classes

- Em UML



Classes

- Em Java

```
class Empregado
{
    private String nome;
    private float salario;
    private static int numeroEmpregados=0;

    public Empregado(String umNome)
    {
        nome = umNome;
        salario = 0.0;
        ++numeroDeEmpregados;
    }
    public void setNome(String umNome)
    {
        nome = umNome;
    }
    public String obtemNome() {
        return nome;
    }
    public void setSalario(float salario){
        this.salario = salario;
    }
    public float obtemSalario(){
        return salario;
    }
}
```

Em C#

```
public class Pessoa {
    // Propriedades da classe
    public string Nome { get; set; }
    public int Idade { get; set; }

    // Construtor da classe
    public Pessoa(string nome, int idade) {
        Nome = nome;
        Idade = idade;
    }

    // Método da classe
    public void Apresentar() {
        Console.WriteLine("Olá, meu nome é " +
Nome + " e eu tenho " + Idade + " anos.");
    }
}
```

Classes

```
#include <iostream>
#include <string>

class Pessoa {
private:
    std::string nome;
    int idade;

public:
    // Construtor
    Pessoa(std::string nome, int idade) : nome(nome), idade(idade) { }
    // Método para obter o nome da pessoa
    std::string getNome() const {
        return nome;
    }
    // Método para obter a idade da pessoa
    int getIdade() const { ←
        return idade;
    }
    // Método para imprimir informações da pessoa
    void mostrarInfo() const {
        std::cout << "Nome: " << nome << "\nIdade: " << idade << std::endl;
    }
};

int main() {
    // Criando uma instância da classe Pessoa
    Pessoa pessoal("João", 25);
    // Usando os métodos da classe para obter informações e mostrar na tela
    std::cout << "Nome: " << pessoal.getNome() << std::endl;
    std::cout << "Idade: " << pessoal.getIdade() << std::endl;
    // Usando o método para mostrar informações completas
    pessoal.mostrarInfo();
    return 0;
}
```

Em C++

Indica que a função não modifica o estado do objeto em que é chamada. Não é obrigatório.

Classes

Classes: definição

- A declaração de uma classe é composta por:
 - **Modificador de acesso:** indica a visibilidade da classe (public, protected, private)
 - **Identificação:** o nome da classe
 - **Herança:** o nome da superclasse, se houver, precedida de: ::
 - Interfaces: a lista das interfaces implementadas (caso haja), separadas por vírgulas, precedida pela palavra-chave implements
 - **Corpo da classe:** envolto por chaves {}, contém os atributos, construtores e métodos da classe.

Definição de classes

Modificador de
Visibilidade

```
class Pessoa {  
private:  
    std::string nome;    // Atributo privado  
    int idade;          // Atributo privado  
  
public:  
    // Construtor  
    Pessoa(std::string nome, int idade) : nome(nome),  
    idade(idade) {}  
  
    // Métodos públicos  
    void apresentar() {  
        std::cout << "Meu nome é " << nome << " e  
        tenho " << idade << " anos." << std::endl;  
    }  
  
protected:  
    // Membros protegidos (se houver)  
};
```

Propriedade

Construtor

Definição de classes (.h)

```
#ifndef CARRO_H
#define CARRO_H

#include <string>

class Carro {
private:
    std::string marca;
    std::string modelo;
    int ano;

public:
    // Construtor
    Carro(std::string marca, std::string modelo, int ano);

    // Métodos para obter informações
    std::string getMarca() const;
    std::string getModelo() const;
    int getAno() const;

    // Método para mostrar detalhes do carro
    void mostrarDetalhes() const;
};

#endif // CARRO_H
```

Definição de classes (.cpp)

```
#include "Carro.h"
#include <iostream>

// Definição do construtor
Carro::Carro(std::string marca, std::string modelo, int ano)
    : marca(marca), modelo(modelo), ano(ano) {}

// Implementação dos métodos para obter informações
std::string Carro::getMarca() const {
    return marca;
}

std::string Carro::getModelo() const {
    return modelo;
}

int Carro::getAno() const {
    return ano;
}

// Implementação do método para mostrar detalhes do carro
void Carro::mostrarDetalhes() const {
    std::cout << "Marca: " << marca << "\nModelo: " << modelo << "\nAno: " << ano
    << std::endl;
}
```

Atributos de Instância

- Os atributos de instância:
 - Podem ser de qualquer tipo básico da linguagem ou de outra classe
 - Pode ser um vetor de qualquer tipo básico ou mesmo um vetor de objetos
 - **Não são automaticamente inicializados pelo compilador**
 - São copiados para todos os objetos

```
class Pessoa {  
public:  
    std::string nome; // Atributo de instância  
    int idade;       // Atributo de instância  
};  
  
int main() {  
    Pessoa pessoal; // Criando uma instância da classe Pessoa  
    pessoal.nome = "João";  
    pessoal.idade = 30;  
    Pessoa pessoa2; // Criando outra instância da classe Pessoa  
    pessoa2.nome = "Maria";  
    pessoa2.idade = 25;  
    // Cada instância tem seus próprios valores exclusivos  
    std::cout << pessoal.nome << " tem " << pessoal.idade << " anos." << std::endl;  
    std::cout << pessoa2.nome << " tem " << pessoa2.idade << " anos." << std::endl;  
    return 0;  
}
```

Atributos de Instância

- São declarados no formato:

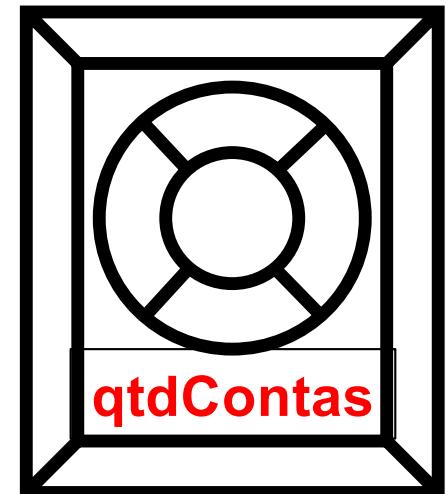
```
<modificador> tipoAtributo nomeAtributo;
```

- Onde:

- **modificador**: indica a visibilidade da variável (public, protected, private)
- **tipoAtributo**: o tipo (primitivo ou por referência) da variável
- **nomeAtributo**: o identificador da variável

Variável de classe

- Uma classe pode ter variáveis contendo informações úteis, como:
 - número de objetos instanciados da classe até certo instante
 - valor médio de determinada propriedade
- Essas **variáveis não** devem ser criadas para cada objeto
 - Para cada classe existirá apenas uma única cópia de cada variável: variável de classe ou atributo de classe
 - A variável de classe existe a partir do ponto em que a classe que a define é carregada na memória



Variável de classe

- As variáveis de classe:

- Podem ser de qualquer tipo básico ou de referência
- NÃO são copiados para todos os objetos
- São declarados no formato:

```
<modificador> static tipoAtributo nomeAtributoClasse;
```

- Onde:
 - **modificador**: indica a visibilidade da variável (public, protected, private)
 - **tipoAtributo**: o tipo (primitivo ou por referência) da variável
 - **nomeAtributoClasse**: o identificador da variável

Variável de classe

- Esta variável é da classe e não de cada objeto.

Contador: 2

```
#include <iostream>

class MinhaClasse {
public:
    static int contador; // Variável de classe

    MinhaClasse() {
        contador++; // Incrementa o contador ao criar um objeto
    }
};

int MinhaClasse::contador = 0; // Inicialização da variável de classe

int main() {
    MinhaClasse objeto1;
    MinhaClasse objeto2;

    std::cout << "Contador: " << MinhaClasse::contador << std::endl; // Acesso à variável de classe

    return 0;
}
```

Atributo de instância x de classe

- Considere a classe Quadrado com um único **atributo de classe**: cor

```
#include <iostream>
#include <string> // Importante para usar std::string

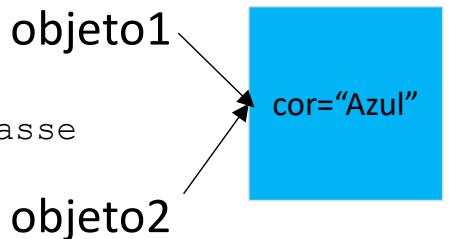
class Quadrado {
public:
    static std::string cor; // Variável de classe
};

std::string Quadrado::cor = "Azul"; // Inicialização da variável de classe

int main() {
    Quadrado objeto1;
    Quadrado objeto2;

    std::cout << "Cor do objeto1: " << objeto1.cor << std::endl; // Acesso à variável de classe
    std::cout << "Cor do objeto2: " << objeto2.cor << std::endl; // Acesso à variável de classe
    std::cout << "Cor de qualquer objeto: " << Quadrado::cor << std::endl; // Acesso à variável de classe
    return 0;
}
```

Cor do objeto1: Azul
Cor do objeto2: Azul
Cor de qualquer objeto: Azul



Atributo de instância x de classe

- Considere a classe Quadrado com um único **atributo de instância**: cor

```
#include <iostream>
#include <string> // Importante para usar std::string

class Quadrado {
public:
    std::string cor; // Variável de instância
    Quadrado(std::string cor): cor(cor) {};
    void setCor(std::string cor) {
        this->cor = cor;
    }
};

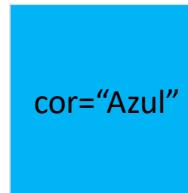
int main() {
    Quadrado objeto1("Azul");
    Quadrado objeto2("Vermelho");

    std::cout << "Cor do objeto1: " << objeto1.cor << std::endl;
    std::cout << "Cor do objeto2: " << objeto2.cor << std::endl;

    return 0;
}
```

Cor do objeto1: Azul
Cor do objeto2: Vermelho

objeto1



objeto2



Atributo de instância x de classe

- Considere a classe Quadrado com um único **atributo de instância**: cor

```
#include <iostream>
#include <string> // Importante para usar std::string

class Quadrado {
public:
    std::string cor; // Variável de instância
    Quadrado(std::string cor): cor(cor) {}
    void setCor(std::string cor) {
        this->cor = cor;
    }
};

int main() {
    Quadrado objeto1("Azul");
    Quadrado objeto2("Vermelho");

    std::cout << "Cor do objeto1: " << objeto1.cor << std::endl;
    std::cout << "Cor do objeto2: " << objeto2.cor << std::endl;
    objeto1.setCor("Verde");
    std::cout << "Cor do objeto1: " << objeto1.cor << std::endl;
    std::cout << "Cor do objeto2: " << objeto2.cor << std::endl;

    return 0;
}
```

Cor do objeto1: Azul
Cor do objeto2: Vermelho
Cor do objeto1: Verde
Cor do objeto2: Vermelho

objeto1



objeto2



Métodos

- Um **método** implementa algum aspecto do comportamento do objeto
- Um **método** é uma subrotina definida na classe que pode acessar o estado interno de um objeto para realizar alguma operação
- Um **método** é uma sequência de ações, executada por um objeto, que pode alterar ou verificar seu estado (valor dos atributos)
- Enquanto o valor dos atributos (de instância) reside no objeto, o método reside na classe

Métodos

- **Assinatura de um método:**
 - **Modificadores:** tratam da visibilidade do método (public, protected, private) e se ele não pertence ao objeto, mas à classe (static)
 - **Tipo de retorno:** o tipo de dado (primitivo ou por referência) que o método deverá retornar, ou void se o método não retorna informação
 - **Identificador do método:** nome do método
 - **Lista de parâmetros:** lista de argumentos do método;
- **Corpo do método:**
 - Contém a implementação do método
- **Mensagens e Métodos**
 - Mensagem é o conceito
 - Método é a implementação

Métodos

- Assinatura de um método:

```
class MinhaClasse {  
public:  
    // Assinatura do método  
    int multiplicar(int a, int b);  
};
```

- Implementação:

```
#include "MinhaClasse.h"  
  
// Implementação do método  
int MinhaClasse::multiplicar(int a, int b) {  
    return a * b;  
}
```

Construtores e Destrutores

- Algumas linguagens implementam métodos especiais:
 - **Construtores:** usados para criar e inicializar objetos novos
 - Em C++, C# e Java ganham o mesmo nome da classe
 - Pode-se ter várias versões de construtores para cada classe, mudando a quantidade e o tipo de parâmetros
 - **Destrutores:** usados para destruir objetos
 - Em C++ e C# ganham o nome da classe precedido de um ~ (til) e não recebem parâmetros.
 - Java não implementa o conceito de destrutor e faz a coleta de lixo automática. C# também o faz

Construtores e Destrutores

- **Construtor**

- Possuem o mesmo nome da classe
- Devem ser públicos
- Não tem tipo de retorno
- É um método especial da classe responsável pela inicialização de um objeto
- Possui inicializadores de Membros:
- Se não for explicitamente declarado um construtor, o compilador fornece um construtor padrão sem parâmetros
 - Invocado apenas 1 vez por objeto criado
- Atributos membros constantes devem ser inicializados em um construtor de inicialização, uma vez que não podem ser modificados após a inicialização.

Construtores e Destrutores

- **Construtor**

- **Chamada de Construtores de Classe Base:** Quando uma classe deriva de outra classe, o construtor da classe base é chamado antes do construtor da classe derivada, garantindo que os atributos da classe base sejam inicializados antes dos atributos da classe derivada.
- **Delegação de Construtores:** Em C++11 e posteriores, você pode usar a delegação de construtores para chamar um construtor a partir de outro construtor da mesma classe.

Construtores e Destruidores

```
#include <iostream>
#include <string>

class Pessoa {
private:
    std::string nome;
    int idade;

public:
    // Construtor com um parâmetro (delegando para o construtor de 2 parâmetros)
    Pessoa(std::string n) : Pessoa(n, 0) {}

    // Construtor com dois parâmetros
    Pessoa(std::string n, int i) : nome(n), idade(i) {}

    // Destrutor
    ~Pessoa() { std::cout << "Destrutor chamado para " << nome << std::endl; }

    void mostrarInfo() {
        std::cout << "Nome: " << nome << "\nIdade: " << idade << std::endl;
    }
};

int main() {
    Pessoa pessoa1("Alice");
    Pessoa pessoa2("Bob", 25);
    pessoa1.mostrarInfo();
    pessoa2.mostrarInfo();
    return 0;
}
```

Construtores e Destruidores

- Destruidores são úteis para realizar tarefas de limpeza e liberação de recursos
- São chamados quando um objeto sai do escopo ou é explicitamente deletado
- O destrutor é automaticamente chamado quando um objeto é destruído, **MAS ELE NÃO** é responsável por liberar a memória alocada dinamicamente

Métodos de Acesso

- São métodos criados para permitir que atributos protegidos possam ter seu valores lidos e modificados por elementos de outras classes
- No geral, são distribuídos em dois grupos:
 - **Sets**: permitem que o atributo em questão seja alterado
 - **Gets**: permite que o valor corrente do atributo seja lido

Métodos de Acesso

```
#include <iostream>
#include <string>

class Pessoa {
private:
    std::string nome;
    int idade;
public:
    // Construtor
    Pessoa(std::string n, int i) : nome(n), idade(i) {}
    // Método de acesso para obter o nome
    std::string getNome() const {
        return nome;
    }
    // Método de acesso para obter a idade
    int getIdade() const {
        return idade;
    }
    // Método de modificação para definir a idade
    void setIdade(int novaIdade) {
        if (novaIdade >= 0) {
            idade = novaIdade;
        }
    }
};

int main() {
    Pessoa pessoa("Alice", 30);
    std::cout << "Nome: " << pessoa.getNome() << "\nIdade: " << pessoa.getIdade() << std::endl;
    pessoa.setIdade(31); // Alterando a idade usando o método setter
    std::cout << "Nova idade: " << pessoa.getIdade() << std::endl;
    return 0;
}
```

Sobrecarga de Métodos

- Podemos ter métodos com o mesmo nome desde que tenham assinaturas diferentes. Quando isso acontece, dizemos que temos sobrecarga de métodos (overloading).

```
#include <iostream>

class Calculadora {
public:
    int somar(int a, int b) {
        return a + b;
    }

    double somar(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculadora calc;

    int resultadoInt = calc.somar(5, 3);
    std::cout << "Resultado da soma (int): " << resultadoInt << std::endl;

    double resultadoDouble = calc.somar(2.5, 3.7);
    std::cout << "Resultado da soma (double): " << resultadoDouble << std::endl;

    return 0;
}
```

Sobrecarga de Métodos

```
#include <iostream>

class Calculadora {
public:
    // Método para somar dois números
    int somar(int a, int b) {
        return a + b;
    }

    // Sobrecarga para somar três números
    int somar(int a, int b, int c) {
        return a + b + c;
    }
    // Sobrecarga com parâmetros dobrados
    int somar(int num, ...) {
        int resultado = num;
        va_list args;
        va_start(args, num);
        int val;
        while ((val = va_arg(args, int)) != 0) {
            resultado += val;
        }
        va_end(args);
        return resultado;
    }
};

int main() {
    Calculadora calculadora;
    std::cout << "Soma (2 parâmetros): " << calculadora.somar(5, 3) << std::endl;
    std::cout << "Soma (3 parâmetros): " << calculadora.somar(5, 3, 7) << std::endl;
    std::cout << "Soma (parâmetros dobrados): " << calculadora.somar(5, 3, 7, 2, 10, 0) << std::endl;
    return 0;
}
```

Com dobra

Sobrecarga de Operadores

- A sobrecarga de operadores é feita definindo funções membro ou funções globais especiais que indicam como um operador deve se comportar quando usado com objetos da classe.
- Essas funções têm nomes especiais, começando com a palavra-chave `operator` seguida pelo operador que você deseja sobrestrar.

```
#include <iostream>
class Complexo {
public:
    Complexo(double r, double i) : real(r), imaginario(i) {}
    Complexo operator+(const Complexo &outro) {
        return Complexo(real + outro.real, imaginario + outro.imaginario);
    }
    void mostrar() {
        std::cout << real << " + " << imaginario << "i" << std::endl;
    }
private:
    double real;
    double imaginario;
};
int main() {
    Complexo c1(2.0, 3.0);
    Complexo c2(1.0, 2.0);
    Complexo resultado = c1 + c2;
    resultado.mostrar();
    return 0;
}
```

C++: Instanciando Objetos

Estaticamente

```
public class Pessoa
{
    // Atributos de instância
    public string Nome { get; set; }
    public int Idade { get; set; }

    // Método de instância
    public void Falar()
    {
        Console.WriteLine("Oi, meu nome é " + Nome + " e eu tenho " + Idade + " anos.");
    }
}

// Utilizando a classe Pessoa
static void Main(string[] args)
{
    // Criando um objeto da classe Pessoa
    Pessoa pessoal = new Pessoa();
    pessoal.Nome = "João";
    pessoal.Idade = 30;
    pessoal.Falar();

    // Criando outro objeto da classe Pessoa
    Pessoa pessoa2 = new Pessoa();
    pessoa2.Nome = "Maria";
    pessoa2.Idade = 25;
    pessoa2.Falar();
}
```

C++: Instanciando Objetos

Dinamicamente

```
#include <iostream>

class Exemplo {
public:
    Exemplo() {
        std::cout << "Objeto Exemplo criado!" << std::endl;
    }

    ~Exemplo() {
        std::cout << "Objeto Exemplo destruído!" << std::endl;
    }

    void MetodoExemplo() {
        std::cout << "MétodoExemplo() foi invocado!" << std::endl;
    }
};

int main() {
    // Criando um objeto Exemplo dinamicamente
    Exemplo *ptr = new Exemplo();

    // Invocando o método do objeto
    ptr->MetodoExemplo();

    // Liberando a memória alocada
    delete ptr;

    return 0;
}
```

- Na próxima aula
 - Variáveis e Métodos
 - Relacionamentos: herança