

CONCEITOS BÁSICOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS

Algoritmos e Estruturas de Dados

Daniel Guerreiro e Silva



Universidade de Brasília

ROTEIRO

1. Objeto e Classe
2. Atributos e Métodos (membros de uma classe)
3. Herança
 - Sobrescrita (*overriding*)
4. Polimorfismo
 - Sobrecarga (*overloading*)

Leitura sugerida: Cap. 1 do livro-texto, seções 1.1-1.3, 1.4, 1.4.1, 1.6, 1.7, 1.8, 1.9.

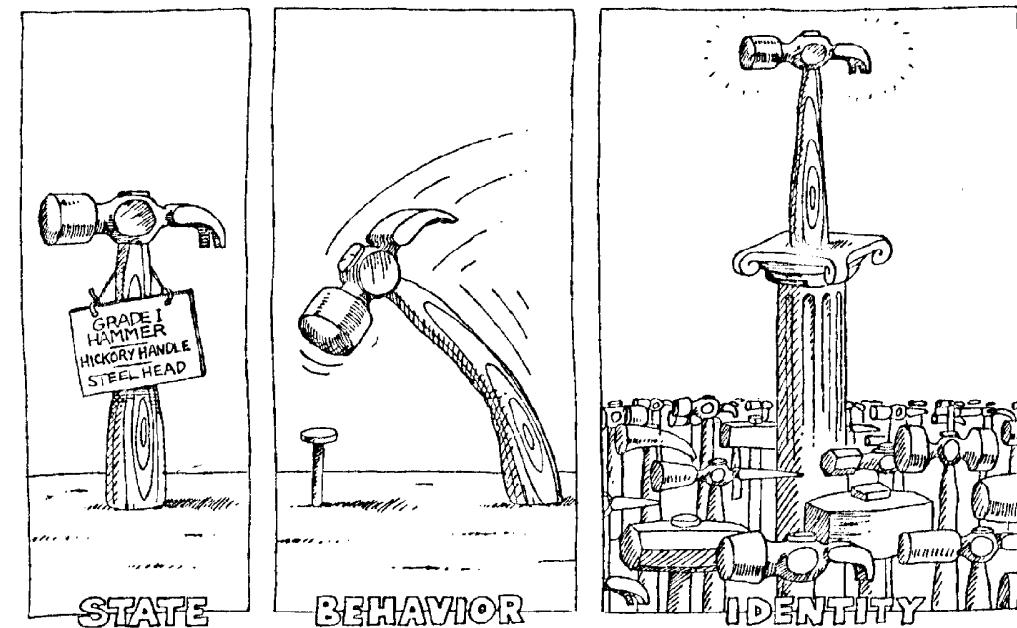
OBJETO E CLASSE

OBJETOS “REAIS”

- Olhe os objetos ao seu redor: eles possuem **atributos, estado, comportamento e identidade**
 - Em quais estados ele pode estar?
 - Que comportamentos ele pode assumir?
- Objetos do mundo real podem variar em complexidade
 - número de estados, número de comportamentos
 - Podem conter outros objetos

As mesmas ideias podem ser associadas **a objetos de software!**

Classes and Objects



An object has state, exhibits some well-defined behavior, and has a unique identity.

OBJETOS...

- Os **atributos** de um objeto definem as qualidades e características da entidade que ele representa
- O **estado** de um objeto é o particular conjunto de valores de seus atributos, em um dado momento
- O **comportamento** de um objeto é definido pelas alterações do seu estado em resposta a mensagens que ele recebe (interação com outros objetos)

CLASSES E OBJETOS

Classe

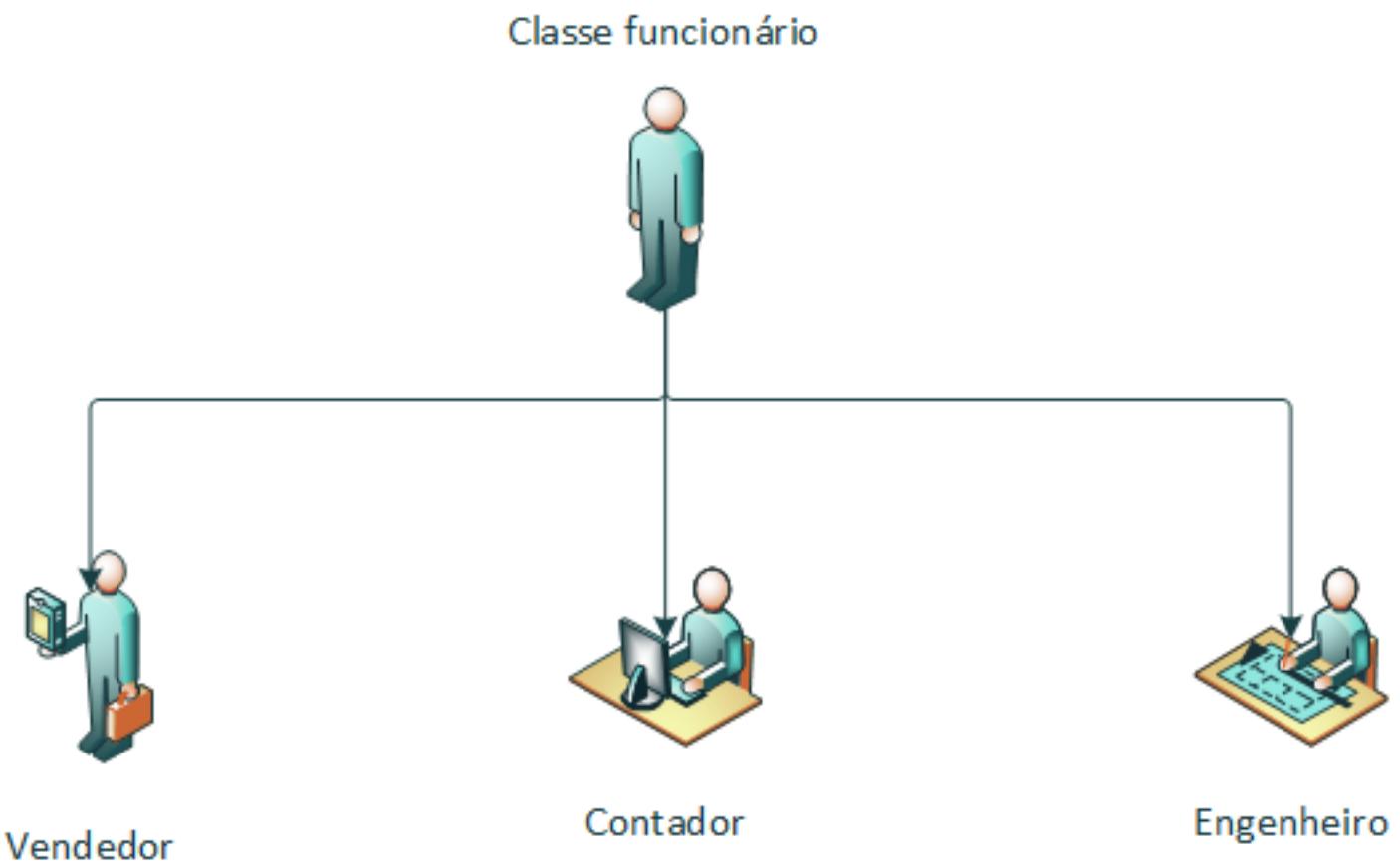
- “esqueleto” que serve de molde para a criação de objetos. Define os atributos e os comportamentos que são comuns a todos os objetos da classe

Objeto

- instância de uma classe, ou seja, um elemento de programação que apresenta os comportamentos e atributos definidos pela classe da qual foi instanciado

CLASSES

- Definem elementos de mesma natureza
- Modelam características comuns



CLASSES EM PROGRAMAS DE COMPUTADOR

- Em um programa orientado a objetos, escrevemos (programamos) **definições de classes** em vez de subrotinas.
- Cada instância (objeto) possui o seu próprio conjunto de valores para os atributos, independentemente de outras instâncias da mesma ou de outras classes
- Porém, todas as instâncias de uma mesma classe **compartilham as mesmas definições** de métodos

DESIGN ORIENTADO A OBJETOS

- Modela software em termos semelhantes àqueles que as pessoas usam para descrever objetos da vida real
 - Tira proveito dos relacionamentos de **classe** e de **herança**
- Modela-se através de uma linguagem gráfica, a **UML**, para chegar a um **design** que atenda os requisitos do cliente
- Java, Python, C++ são exemplos de linguagens orientadas a objetos

VANTAGENS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

- Gerenciar a complexidade dos sistemas que são desenvolvidos nas organizações, viabilizando o trabalho conjunto de grandes equipes
- Aumentar a produtividade dos analistas e programadores pela reutilização de código pronto e depurado escrito em sistemas anteriores ou adquiridos no mercado

CLASSES E OBJETOS EM C++

Para criar uma classe em C++, use a palavra reservada **class**

- Alternativamente, pode-se usar a palavra reservada **struct**

```
class nome_classe {  
    especificador_acesso_1:  
        membro11;  
        membro21;  
        ...  
    especificador_acesso_2:  
        membro12;  
        membro22;  
        ...  
    ...  
};
```

CLASSES E OBJETOS EM C++

```
class Arvore{  
    int anoplantacao; //isto é um atributo  
    string nome; //isto é outro atributo  
  
public:  
    //isto é um construtor de classe  
    Arvore(int n1, string n2){  
        anoplantacao = n1;  
        nome = n2;  
    }  
  
    //isto é uma função-membro da classe (método)  
    void meu_nome(int anocorrente){  
        cout << "Meu nome e: " << nome << " e tenho "  
        << anocorrente-anoplantacao << " anos.\n";  
    }  
};
```

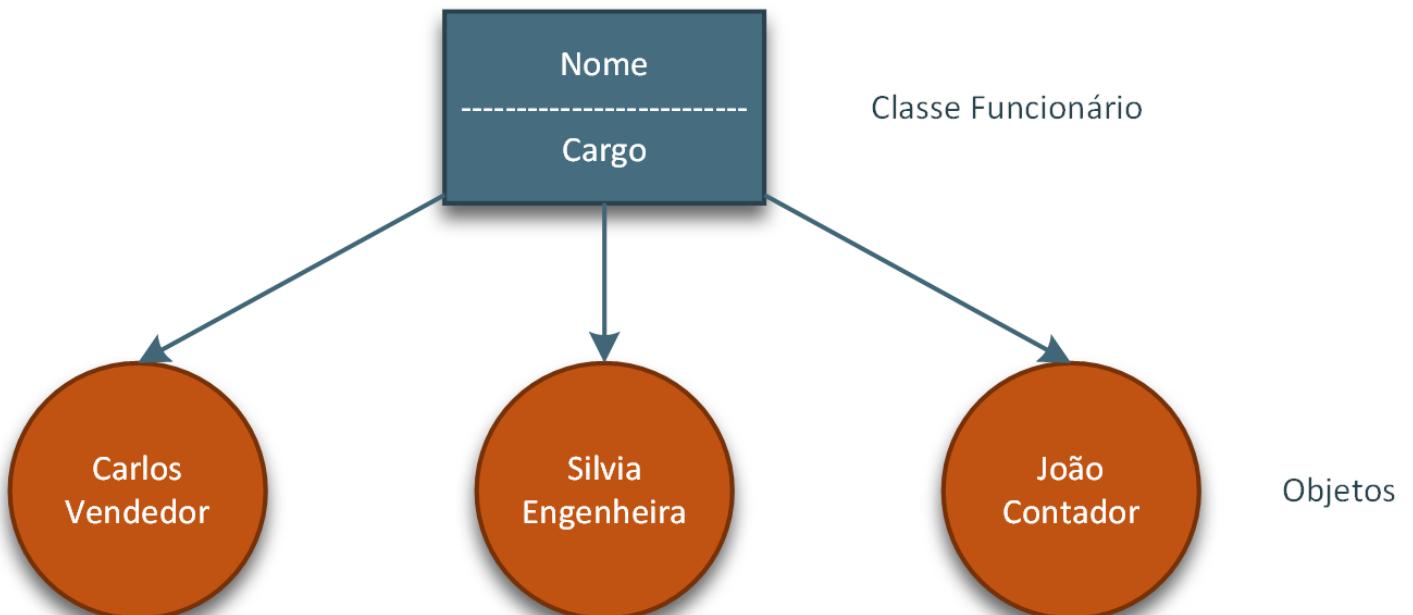
ATRIBUTOS E MÉTODOS (MEMBROS DE UMA CLASSE)

INSTÂNCIAS...

- Uma classe **descreve** os elementos que compõem um objeto
- Importante: a classe é usada apenas para modelar as características de um conjunto de objetos
- A criação ou **instanciação** de um objeto de uma classe provoca a alocação de memória para armazenar os seus atributos

INSTÂNCIAS...

- Objetos são instâncias de classes
- Os atributos e métodos de uma classe tornam-se disponíveis em cada objeto instanciado



CLASSES E OBJETOS EM C++

Um objeto é **instanciado** em C++ ao declarar uma variável do tipo daquela classe:

```
Arvore arv(2009, "Laranjeira");
```

Variável que
referencia o objeto

Argumentos para
o construtor

INSTÂNCIAS...

Sintaxe

nome_classe nome_objeto(<<parâmetros construtor>>);

ou

nome_classe nome_objeto();

ou

nome_classe nome_objeto;

nome_objeto é a variável que referencia o objeto criado nesta declaração.

- Após a instanciação, **os membros** (atributos e funções) públicos do objeto podem ser acessados pelo operador ponto (.), similar ao acesso de campos de uma variável registro (struct):

```
nome_objeto.nome_membro;
```

VARIÁVEIS DE INSTÂNCIA

- Os atributos dos objetos são definidos por **variáveis**.
- As **variáveis de instância** definem atributos que podem receber diferentes valores em objetos distintos.

VARIÁVEIS DE INSTÂNCIA

Exemplo

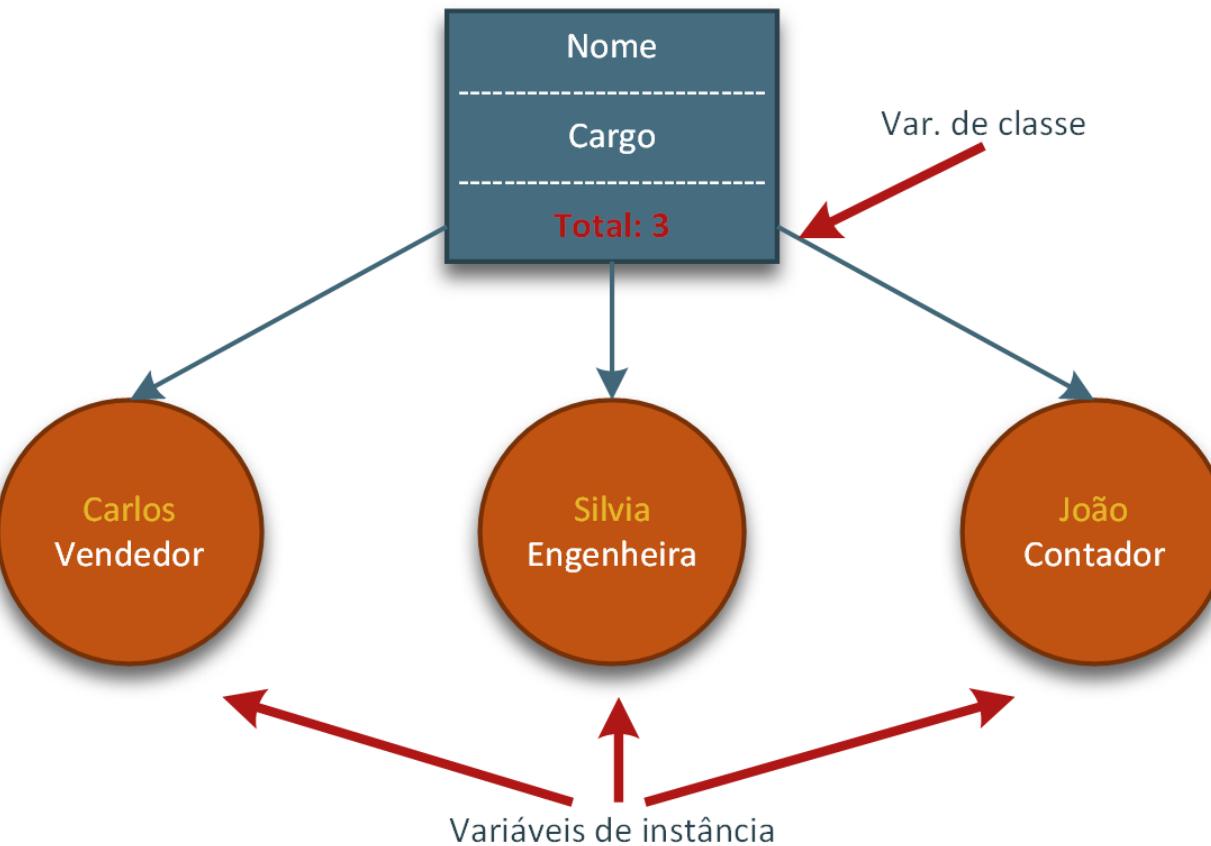
- na classe funcionário, cada objeto instanciado modela uma pessoa que tem seu próprio nome
- nome é, portanto, uma variável de instância



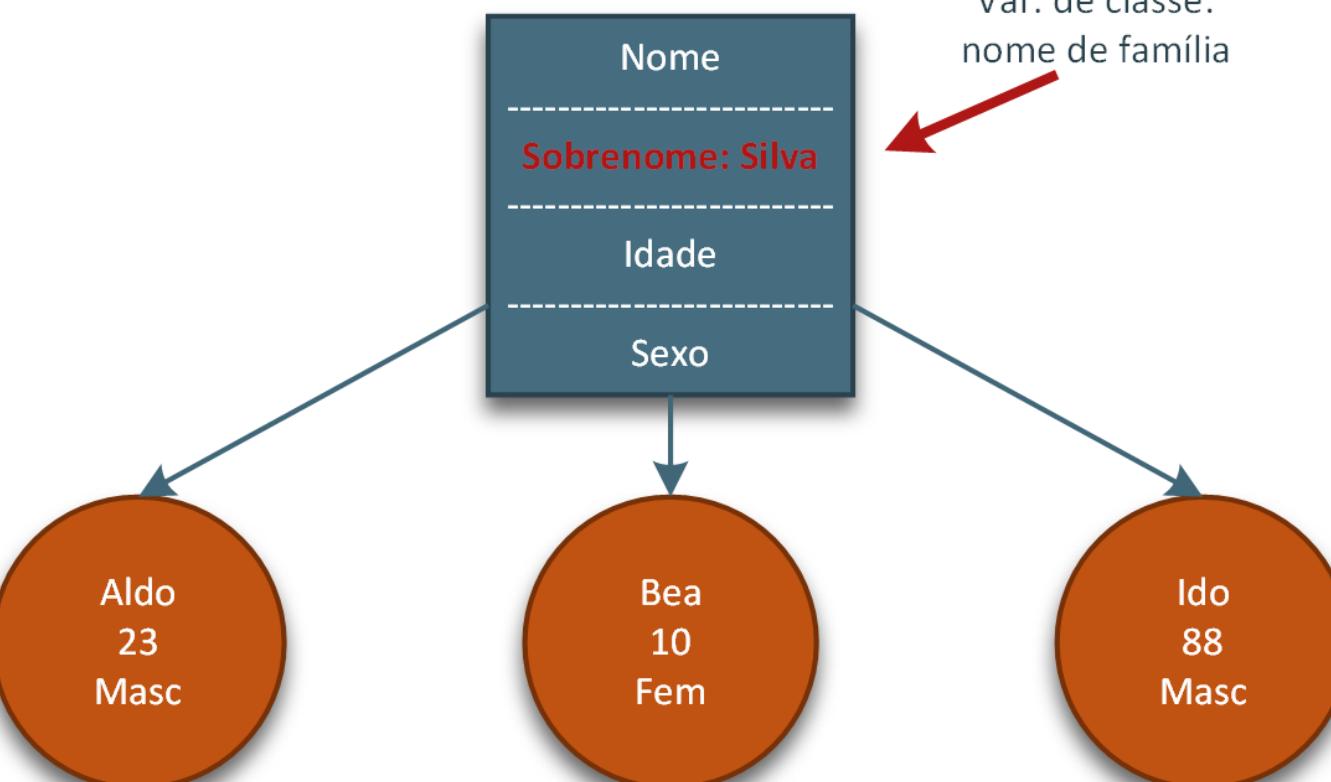
VARIÁVEIS DE CLASSE

- Atributos comuns a todos os elementos da classe são armazenados em **variáveis de classe**
 - Ela armazena informações sobre a classe inteira, não apenas sobre um determinado objeto
- Exemplo:
 - na classe funcionário, o número de funcionários de uma empresa seria uma variável de classe, pois não se refere a nenhum funcionário em particular

CLASSES X INSTÂNCIAS



CLASSES X INSTÂNCIAS



MÉTODOS

- O **comportamento** de um objeto é definido a partir de operações que alteram, em geral, seu estado (os valores de seus atributos)
- Em Orientação a Objetos, estes procedimentos são chamados de **métodos ou funções-membro**

MÉTODOS...

- Objetos comunicam-se entre si através de métodos
- Pode-se escrever métodos para:
 - consultar os valores de atributos (variáveis de instância) de um objeto
 - alterar o valor de atributos do objeto
 - solicitar um serviço ao objeto
 - fornecer informações ao objeto

MÉTODOS E FUNÇÕES EM C++

- Definição de um método ou de uma função em C++:

```
<tipo_retorno> nomeMetodo(<lista_de_parametros>){  
    <corpo_do_método>  
}
```

- Como dito antes, para invocar um método associado ao objeto, usamos o operador de ponto (.) entre os nomes da variável e do método:

```
nome_objeto.nomeMetodo(<lista_de_argumentos>)
```

EXEMPLO: REPRESENTAÇÃO DE UM RETÂNGULO

- Arquivo retangulo.cpp

```
#include <iostream>
using namespace std;

//declaracao da classe Rectangle
class Rectangle {
    int width, height; //var. de instancia / atributos

public:
    void set_values (int x, int y) { //metodo
        width = x;
        height = y;
    }

    int area () { //metodo
        return width*height;
    }
}; //fim da classe

int main () {

    Rectangle rect, rectb; //criam-se dois objetos

    rect.set_values (3,4); //chamada do metodo para o objeto rect
    rectb.set_values (5,6); //chamada do metodo para o objeto rectb
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;

    return 0;
}
```

UM PROBLEMA

- O que aconteceria no exemplo anterior se invocássemos o método `area` antes de ter invocado `set_values`?
 - Um resultado indeterminado, pois `width` e `height` não tiveram valores atribuídos
- Para se evitar situações como essas que uma classe pode incluir uma função especial chamada de **construtor**

CONSTRUTORES

- Função especial que é automaticamente chamada toda vez que um novo objeto da classe é criado
- Serve para inicializar os membros (atributos) e/ou pré-alocar memória
- O construtor é declarado como uma função-membro qualquer, só que com o nome da própria classe e sem nenhum tipo de retorno, nem mesmo `void`

CONSTRUTORES

```
class minhaClasse{  
    public:  
        minhaClasse(<lista parametros>){  
            <comandos de inicializacao>  
        }  
        //demais metodos, atributos...  
};
```

- Se o programador não escreve o construtor, o compilador cria um construtor padrão (sem parâmetros) que invoca a inicialização padrão de cada membro.

EXEMPLO, AGORA MELHORADO

- Arquivo retangulo2.cpp
- Repare que agora é necessário passar argumentos para o construtor da classe no momento da criação dos dois objetos

```
#include <iostream>
using namespace std;

//declaracao da classe Rectangle
class Rectangle {
    int width, height; //var. de instancia / atributos

public:

    Rectangle(int x, int y) { //construtor da classe
        width = x;
        height = y;
    }

    int area () { //metodo
        return width*height;
    }
};

int main () {

    //criam-se dois objetos, informando os parametros do construtor
    Rectangle rect(3,4), rectb(5,6);

    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;

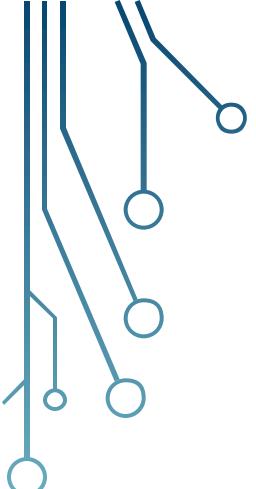
    return 0;
}
```

EXERCÍCIO DE ASSIMILAÇÃO DE CONCEITOS (EAC) 1

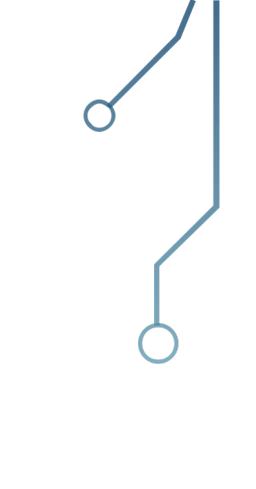
- Vamos escrever um programa `arvore.cpp`, que contém:
 1. Uma classe `Arvore`, que guarda o nome científico e o nome popular de uma árvore. Ela deve conter
 - Um construtor `Arvore::Arvore(std::string, std::string)` que recebe dois parâmetros `string` para inicializar seus respectivos nomes.
 - a função-membro `void Arvore::meu_nome(int)`, que recebe de parâmetro um inteiro e imprime o nome científico se o valor for zero, ou o nome popular caso seja 1
 2. A função `int main()` que cria objetos do tipo `Arvore` e invoca o método `meu_nome` associado a cada um.

HEADER FILES

- Podemos separar as **declarações** de classes e variáveis das **definições (implementações)** de funções-membro e procedimentos.
- Assim podemos
 - modularizar o código em arquivos distintos
 - disponibilizar as declarações (interfaces) para outros programas usuários.
 - desacoplar interfaces de funções / classes das suas implementações
- Arquivos de **declarações** ou *header files* (.hpp ou .h) vs. arquivos de **definições** (.cpp ou .cp)



HEADER FILES

- Recorde-se do exercício arvore.cpp.
 - Por exemplo, podemos reorganizá-lo em
 - um arquivo de declaração (header) arvore.hpp
 - Um arquivo de definição arvore.cpp
 - Um arquivo arvore_main.cpp, com o programa principal que invoca os objetos
 - **Estude estes códigos no Aprender!**
- 

HEADER FILES

- Vamos compilar o exemplo, gerando os arquivos-objeto `arvore.o` e `arvore_main.o` :
`g++ -Wall -std=c++11 -c arvore.cpp`
`g++ -Wall -std=c++11 -c arvore_main.cpp`
- Em seguida, fazemos a ligação (*linking*) dos arquivos-objeto, o que resulta no programa executável `arvore`:
`g++ -Wall -std=c++11 arvore.o arvore_main.o -o arvore`

MODIFICADORES DE ACESSO

- Os modificadores de acesso permitem controlar a forma de acesso (visibilidade) dos membros de uma classe por outras classes ou funções.
- Tal mecanismo permite implementar políticas de encapsulamento (proteção de dados) do código, reduzindo acessos indevidos e brechas de segurança

MODIFICADORES DE ACESSO EM C++

private

- somente são visíveis por outros membros da mesma classe;
- é o modificador padrão de classes declaradas com a palavra-chave `class`

protected

- são visíveis por outros membros da mesma classe e também por membros das suas subclasses

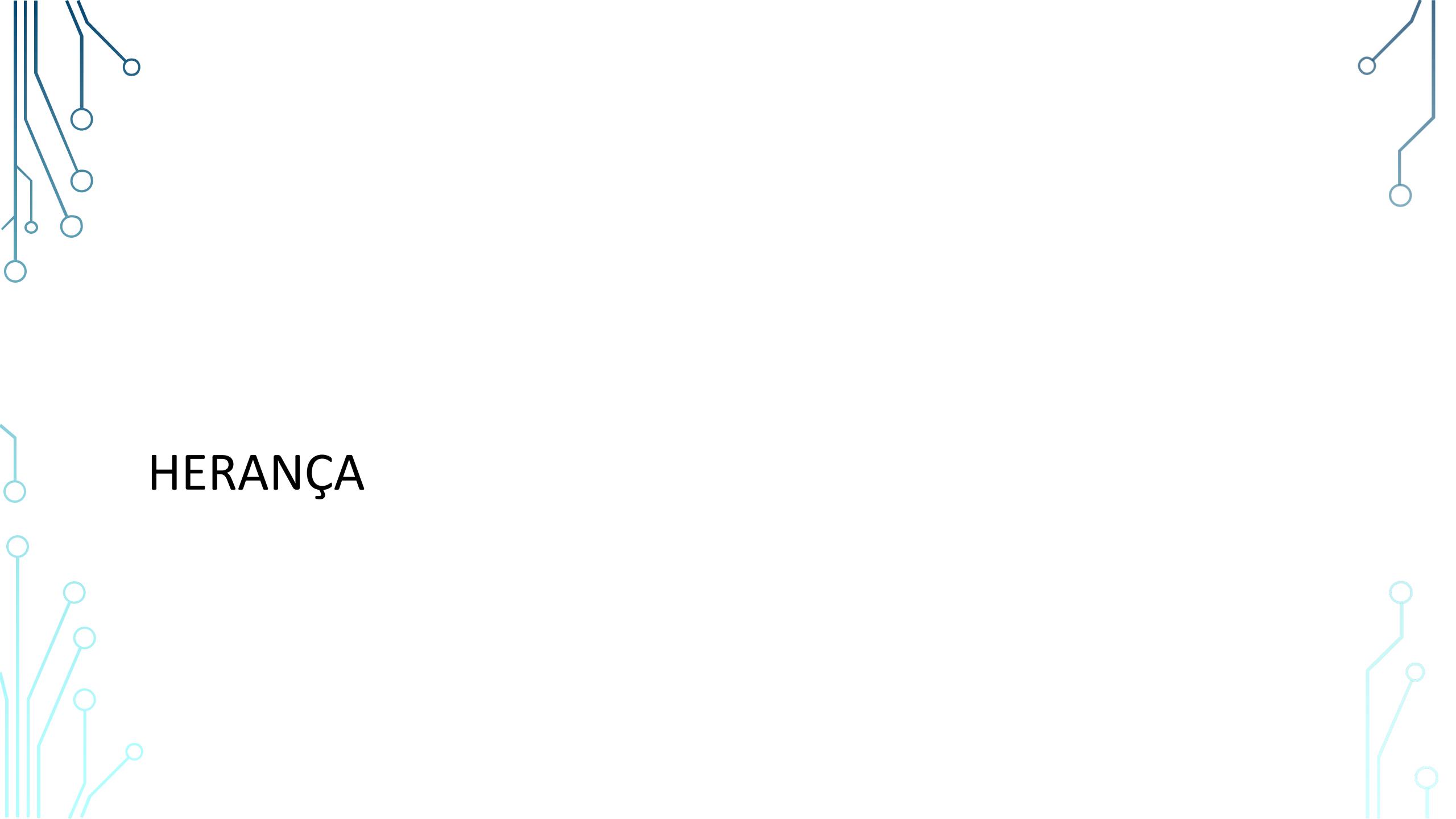
public

- são visíveis em qualquer parte do código em que o objeto seja visível
- é o modificador padrão de classes declaradas com a palavra-chave `struct`

EXEMPLOS

- Estude os arquivos
 - acesso.cpp
 - estudante.cpp e estudante2.cpp

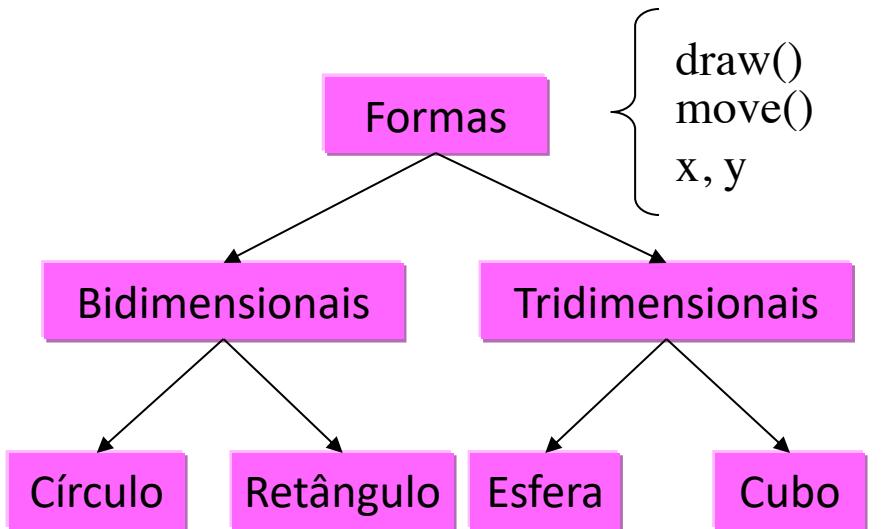
HERANÇA



HERANÇA OU ESPECIALIZAÇÃO

- Uma classe pode ser *derivada* de outra(s) classe(s), e desta forma *herdar* todos os seus membros (atributos, métodos)
- A criação de **subclasses** (ou classes **derivadas**) de uma **superclasse** (ou classe **base**) permite o aumento incremental da funcionalidade dos nossos objetos ou sua **especialização**
- Se precisarmos de um objeto que faça o mesmo que outro objeto faz e “mais alguma coisa”, aproveitamos o código que já está pronto e testado, definindo uma subclasse contendo apenas as novidades

HERANÇA ...



Comportamento e
atributos herdados

HERANÇA SIMPLES EM C++

```
class Base{  
//atributos, métodos...  
};
```

```
class Derivada: public Base{  
//atributos, métodos...  
};
```

HERANÇA MÚLTIPLA EM C++

C++ admite herança **múltipla**: uma classe derivada pode herdar diretamente as características de várias classes-base

```
class Base1{  
//atributos, métodos...  
};
```

```
class Base2{  
//atributos, métodos...  
};
```

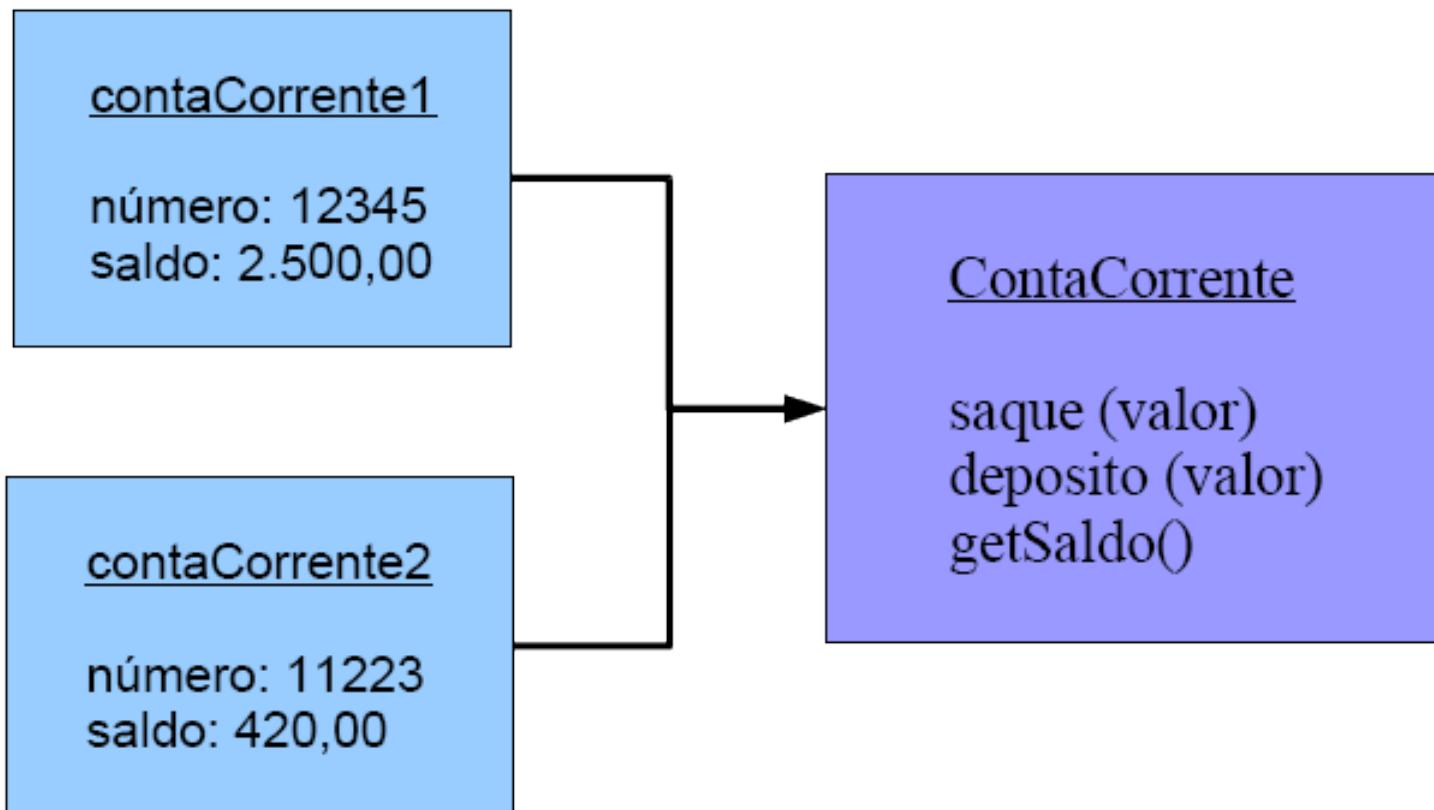
```
class Derivada: public Base1, public Base2{  
//atributos, métodos...  
};
```

CONSTRUTORES E HERANÇA

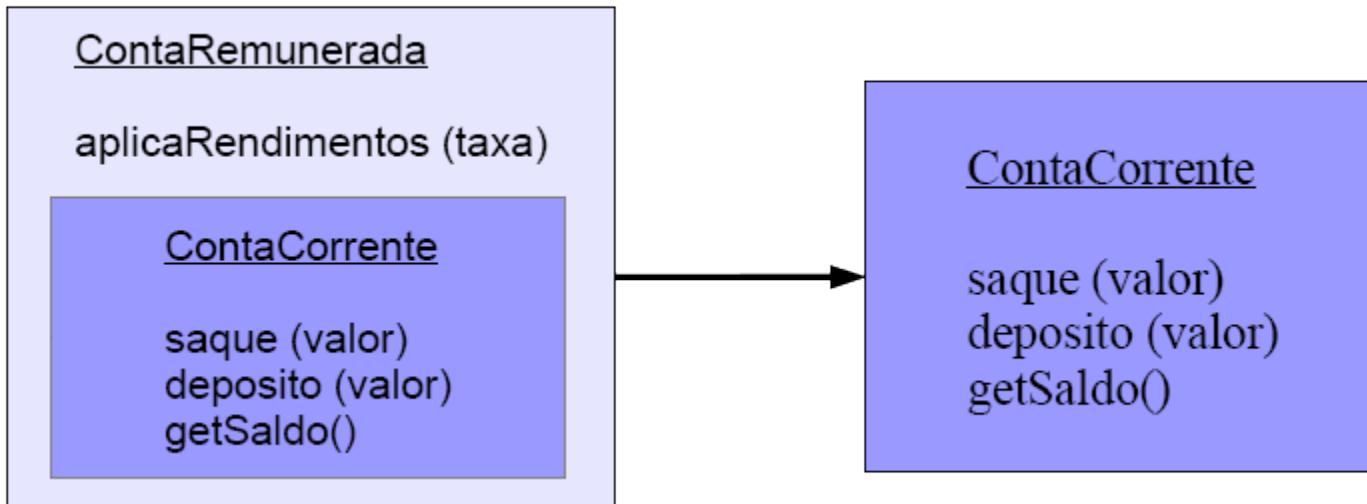
A invocação do construtor da classe base deve aparecer na **lista de inicialização** do construtor da classe derivada.

```
class Pai{  
public:  
    Pai(std::string s) {  
        ...  
    }  
    Pai(int i) {  
        ...  
    }  
};  
  
class Filha: public Pai{  
public:  
    Filha(std::string s): Pai(s) {  
        //outros comandos...  
    }  
    Filha(int i): Pai(i) {  
        //outros comandos...  
    }  
};
```

HERANÇA OU ESPECIALIZAÇÃO - EXEMPLO



HERANÇA OU ESPECIALIZAÇÃO - EXEMPLO



Estude os arquivos conta.hpp, conta.cpp, conta_main.cpp

EXERCÍCIO DE ASSIMILAÇÃO DE CONCEITOS (EAC) 2

1. Escreva a classe `retangulo`, que representa a forma geométrica de um retângulo, tal que possua
 - Dois construtores, o default (sem argumentos) e com os argumentos lado e altura fornecidos
 - os atributos **protected** lado e altura, além de **métodos públicos** do tipo *getters/setters* para cada atributo e método que retorna a área do retângulo
2. Escreva um programa que cria objetos da classe `retangulo` e testa seus valores, calcula a área do objeto associado, mostrando o resultado na saída padrão

EXERCÍCIO DE ASSIMILAÇÃO DE CONCEITOS (EAC) 2

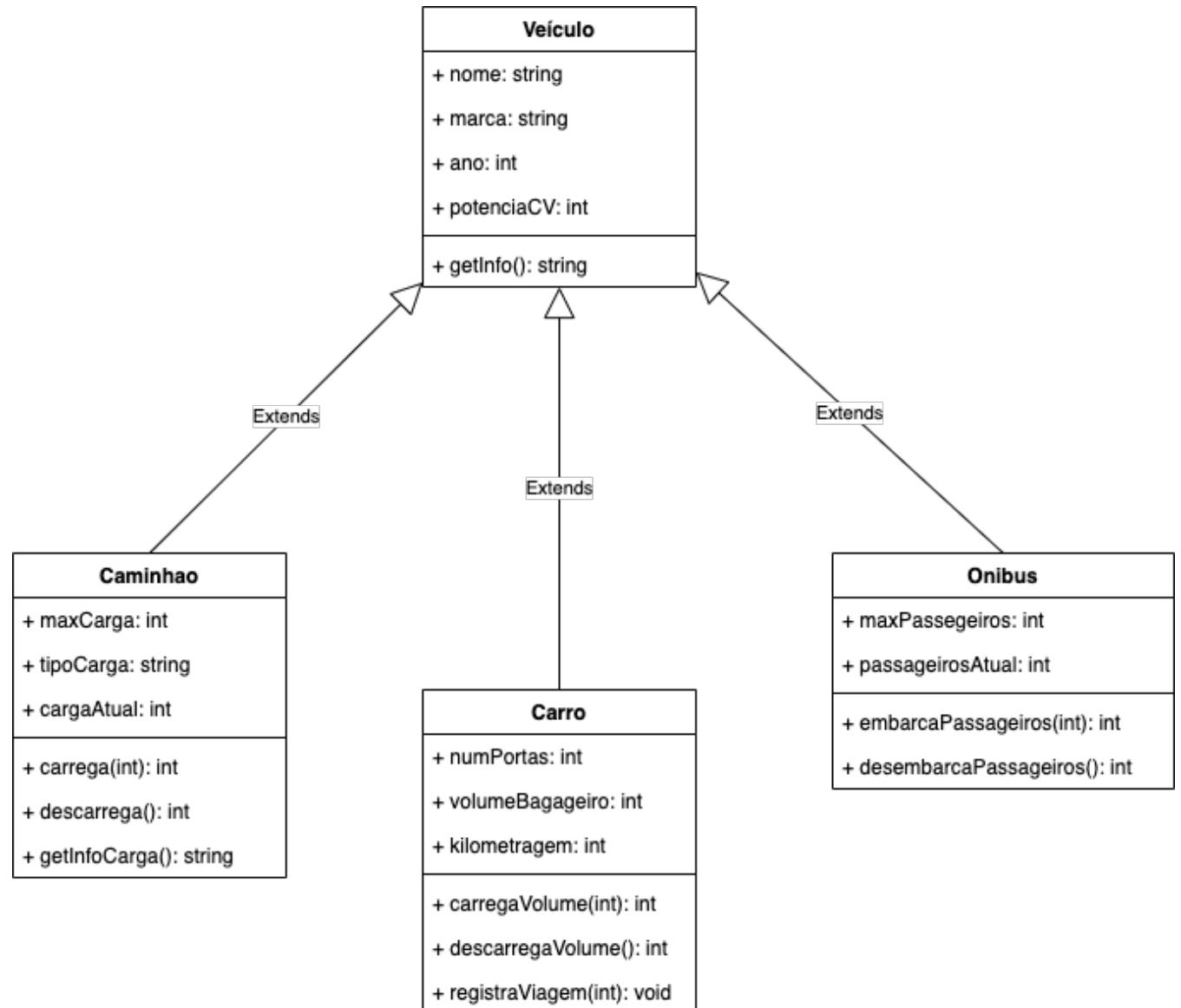
3. Escreva uma outra classe chamada paralelepipedo, **que é classe derivada** de retangulo, e que representa a forma geométrica de um paralelepípedo
 - Deve conter o atributo **privado** profundidade, construtores, getter/setter para a profundidade e método que retorna o volume do paralelepípedo
4. Atualize o programa principal para criar objetos paralelepipedo e testá-los

EXERCÍCIO DE ASSIMILAÇÃO DE CONCEITOS (EAC) 3

- A documentação de um projeto de software orientado a objetos pode envolver diagramas que auxiliem na especificação dos requisitos e funcionalidades do sistema
- Por exemplo, pode-se criar um **diagrama de classes**, o qual serve para modelar os relacionamentos entre as classes de objetos do código a ser escrito.

EXERCÍCIO DE ASSIMILAÇÃO DE CONCEITOS (EAC) 3

- No diagrama de classes ao lado, mostra-se que as classes Caminhao, Carro e Onibus são derivadas da classe base Veiculo.
- Com base no diagrama, implemente em C++ as classes Veiculo e Caminhao, e escreva um pequeno programa que teste as suas funcionalidades.



SOBRESCRITA (OVERRIDING)

- Uma classe derivada pode *sobrepor* métodos herdados, modificando o seu comportamento ou anulando-os completamente
- Declara-se na subclasse o **mesmo nome de método, mesmos parâmetros e mesmo retorno**.
- Se um método é declarado tanto na superclasse quanto na subclasse, o método da subclasse predomina

SOBRESCRITA (*OVERRIDING*)

```
class animal{
public:
    void mover() {
        cout << "Movendo-se..." << endl;
    }
};

class ave: public animal{
public:
    void mover() {
        cout << "Voando..." << endl;
    }
};
```

COMO INVOCAR MÉTODOS SOBRESCRITOS

```
class Circulo {  
public:  
    string toString() {  
        return string("centro = 2,3; raio = 7");  
    }  
};  
  
class CirculoBordado: public Circulo{  
public:  
    string toString() {  
        return Circulo::toString() + "Borda = 1";  
    }  
};
```

SOBRESCRITA (OVERRIDING)

- Estude os exemplos completos nos arquivos animal.cpp e circulos.cpp

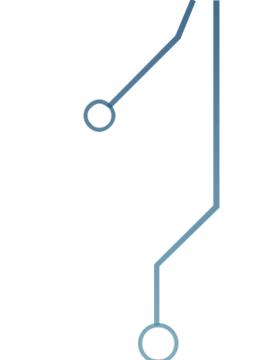
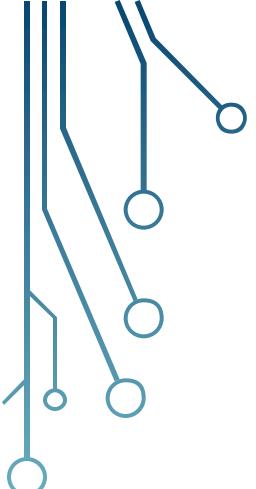
POLIMORFISMO

POLIMORFISMO

Propriedade de se oferecer a mesma interface
para entidades de tipos diferentes.

SOBRECARGA (*OVERLOADING*)

- Definir métodos com o mesmo nome, porém recebendo parâmetros diferentes
- A sobrecarga pode mudar tanto a **quantidade** quanto os **tipos dos parâmetros**; mas não pode **só** mudar o tipo do valor de retorno de um método
- Como já visto antes, é possível (e comum) também fazer sobrecarga de construtores



SOBRECARGA (*OVERLOADING*)

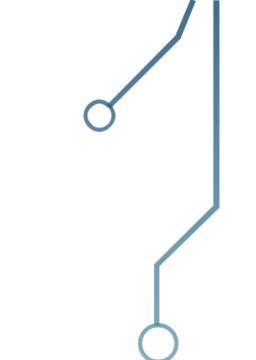
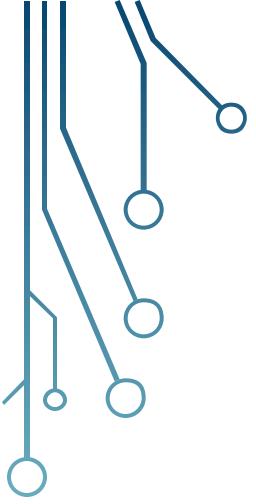
```
double divisao(int n, double d) { ... }
```

```
double divisao(double n, int d) { ... }
```

```
double divisao(double n, double d) { ... }
```

```
double divisao(int n, int d) { ... }
```



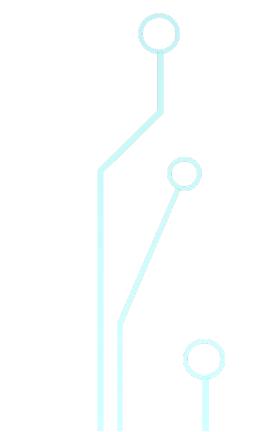


SOBRECARGA (*OVERLOADING*)

```
double media(int data[], int size);
```

```
double media(double data[], int size);
```

```
double media(float data[], int size);
```



EXEMPLO DIRIGIDO

1. Vamos criar uma classe que implementa os métodos sobrecarregados vistos no slide anterior.
2. Vamos escrever um programa para testar um objeto (e seus métodos) criados desta classe.