

Curso de Extensão - INF1900

Programação em C++



Módulo 2

Programação Orientada a Objetos em C++

Aula 2

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.

Variáveis e Métodos

Modularização

- Em C++, há três tipos de módulos:
 - **Namespace:** os namespaces são usados para evitar conflitos de nomes entre diferentes partes do código. Eles podem ser usados para agrupar classes, funções e variáveis relacionadas. Isso ajuda a organizar o código e a manter os nomes únicos. Aqui está um exemplo:
 - **Classes:** são projetos de um objeto, aonde têm características e comportamentos, ou seja, permite armazenar propriedades e métodos dentro dela. Classes **não tem modificadores de visibilidade em C++.**
 - **Métodos:** são serviços implementados na forma de um conjunto de instruções da linguagem (procedimentos algorítmicos) que realizam alguma tarefa específica e podem, como resultado, retornar um valor

Métodos

- **Parâmetros e argumentos**

- Parâmetros referem-se à lista de variáveis declarada em um método
- Argumentos referem-se aos valores que são passados em uma chamada de função.
- Quando um método é chamado, os argumentos precisam ser correspondentes aos tipos e à ordem dos parâmetros declarados

- **Parâmetros por valor**

```
void funcaoPorValor(int x) { x += 10; }
```

- **Parâmetros por referência**

```
void funcaoPorReferencia(int &x) { x += 10; }
```

- **Parâmetros por ponteiro**

```
void funcaoPorPonteiro(int *x) { (*x) += 10; }
```

- **Parâmetros com valor padrão**

```
void funcaoComValorPadrao(int x = 10) { // ... }
```

Métodos: Passagem de Parâmetros

```
#include <iostream>
#include <string>
class Pessoa {
public:
    std::string nome;
    int idade;

    Pessoa(std::string n, int i) : nome(n), idade(i) {}

};

// Função que recebe um objeto Pessoa por valor
void mostrarPessoaPorValor(Pessoa pessoa) {
    std::cout << "Nome: " << pessoa.nome << "\nIdade: " << pessoa.idade << std::endl;
    pessoa.nome = "NovoNome";
}

// Função que recebe um objeto Pessoa por referência
void mostrarPessoaPorReferencia(Pessoa &pessoa) {
    std::cout << "Nome: " << pessoa.nome << "\nIdade: " << pessoa.idade << std::endl;
    pessoa.nome = "NovoNome";
}

int main() {
    Pessoa pessoa1("Alice", 30);
    Pessoa pessoa2("Bob", 25);

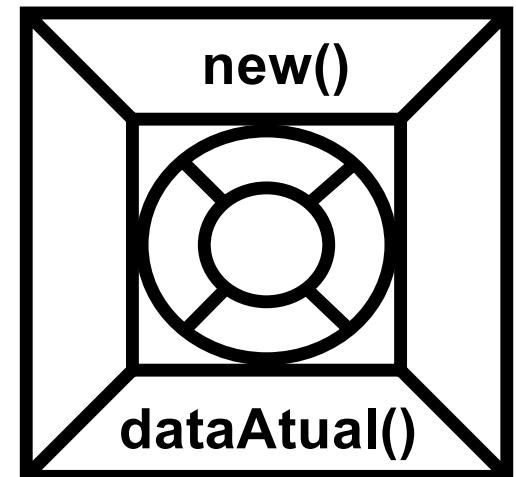
    std::cout << "Passagem por valor:" << std::endl;
    mostrarPessoaPorValor(pessoa1);
    std::cout << "Nome: " << pessoa1.nome << "\nIdade: " << pessoa1.idade << std::endl;

    std::cout << "\nPassagem por referência:" << std::endl;
    mostrarPessoaPorReferencia(pessoa2);
    std::cout << "Nome: " << pessoa2.nome << "\nIdade: " << pessoa2.idade << std::endl;

    return 0;
}
```

Métodos de classe

- Muitas vezes faz-se necessário acessar parte do comportamento de uma classe independentemente de suas instâncias
 - Método `dataAtual` em uma classe `Data`
 - Método `new` que cria um novo objeto da classe
- Neste caso, o comportamento é da classe e não de um objeto em particular
 - É como se a classe tivesse operações
 - próprias
 - Método de classe



Métodos de classe: Declaração e Acesso

- Para declarar um método de classe, usaremos a palavra chave **static**

```
class MinhaClasse {  
public:  
    // Método estático  
    static void metodoEstatico() {  
        // Corpo do método estático  
    }  
};
```

- Por exemplo, um método estático `metodoEstatico()` pertencente a uma classe `MinhaClasse`, pode ser chamado da seguinte forma:

```
MinhaClasse::metodoEstatico([argumentos]);
```

- Se o método `metodoEstatico` for chamado dentro da classe em que foi definido, o nome da classe pode ser omitido.

Métodos de Classe: Exemplo

- Vamos construir uma classe ConversaoDeUnidadesDeTemperatura que contenha métodos estáticos para calcular a conversão entre diferentes escalas de temperatura
 - Considere as fórmulas:
 - De graus Celsius(C) para graus Fahrenheit(F): $F = (9 * C/5) + 32$
 - De graus Fahrenheit (F) para graus Celsius (C): $C = (F - 32) * 5/9$
 - De graus Celsius (C) para graus Kelvin (K): $K = C + 273.15$
 - De graus Kelvin (K) para graus Celsius (C): $C = K - 273.15$
 - De graus Celsius (C) para graus Réaumur (Re): $Re = C * 4/5$
 - De graus Réaumur (Re) para graus Celsius (C): $C = Re * 5/4$
 - De graus Kelvin (K) para graus Rankine (R): $R = K * 1.8$
 - De graus Rankine (R) para graus Kelvin (K): $K = R/1.8$

Métodos de Classe: Exemplo

```
#include <iostream>
class ConversaoDeUnidadesDeTemperatura {
public:
    // De graus Celsius para graus Fahrenheit
    static double CelsiusParaFahrenheit(double celsius) {
        return (9.0 * celsius / 5.0) + 32.0;
    }
    // De graus Fahrenheit para graus Celsius
    static double FahrenheitParaCelsius(double fahrenheit) {
        return (fahrenheit - 32.0) * 5.0 / 9.0;
    }
    // De graus Celsius para graus Kelvin
    static double CelsiusParaKelvin(double celsius) {
        return celsius + 273.15;
    }
    // De graus Kelvin para graus Celsius
    static double KelvinParaCelsius(double kelvin) {
        return kelvin - 273.15;
    }
    // De graus Celsius para graus Réaumur
    static double CelsiusParaReaumur(double celsius) {
        return celsius * 4.0 / 5.0;
    }
    // De graus Réaumur para graus Celsius
    static double ReaumurParaCelsius(double reaumur) {
        return reaumur * 5.0 / 4.0;
    }
    // De graus Kelvin para graus Rankine
    static double KelvinParaRankine(double kelvin) {
        return kelvin * 1.8;
    }
    // De graus Rankine para graus Kelvin
    static double RankineParaKelvin(double rankine) {
        return rankine / 1.8;
    }
};
```

Métodos de Classe: Exemplo

```
int main() {  
    double celsius = 25.0;  
    double fahrenheit = ConversaoDeUnidadesDeTemperatura::CelsiusParaFahrenheit(celsius);  
    std::cout << "Temperatura em Fahrenheit: " << fahrenheit << std::endl;  
  
    double reaumur = ConversaoDeUnidadesDeTemperatura::CelsiusParaReaumur(celsius);  
    std::cout << "Temperatura em Réaumur: " << reaumur << std::endl;  
  
    // ... outros exemplos de conversões  
  
    return 0;  
}
```

Métodos Estáticos: Visibilidade

- Um método estático não consegue enxergar métodos e atributos não estáticos da classe
- Para que um método consiga acessar um método ou atributo não estático da classe, é necessário que o método tenha a referência de um objeto
 - Justificativa: supondo que um método estático chame um método não estático diretamente.
 - Como saber a qual objeto o método estático está se referindo? E se nenhum objeto da classe existir no momento da chamada?

Métodos Estáticos: Visibilidade

- Resumo das visibilidades entre métodos e atributos de instância e de classe
 - Métodos de instância;
 - podem acessar variáveis de instância e métodos de instância diretamente
 - podem acessar variáveis de classe e métodos de classe diretamente
 - Métodos de classe:
 - podem acessar variáveis de classe e métodos de classe diretamente
 - não podem acessar variáveis de instância e métodos de instância diretamente. É necessário o uso de uma referência a um objeto. Além disso, métodos de classe não podem utilizar a palavra-chave `this` dado que não há nenhuma instância associada

Herança

Generalização e Herança

- Relação do tipo generalização/especialização
 - Objetos de um elemento especializado (um filho) podem ser substituídos por objetos do elemento geral (o pai)
 - É uma maneira natural de modelar o mundo real
 - Relação transitiva e anti-simétrica entre classes
 - A modelagem pode ser testada verificando se o elemento especializado “é do tipo” do elemento geral
- Generalização
 - nome do relacionamento
- Herança
 - mecanismo que implementa o relacionamento

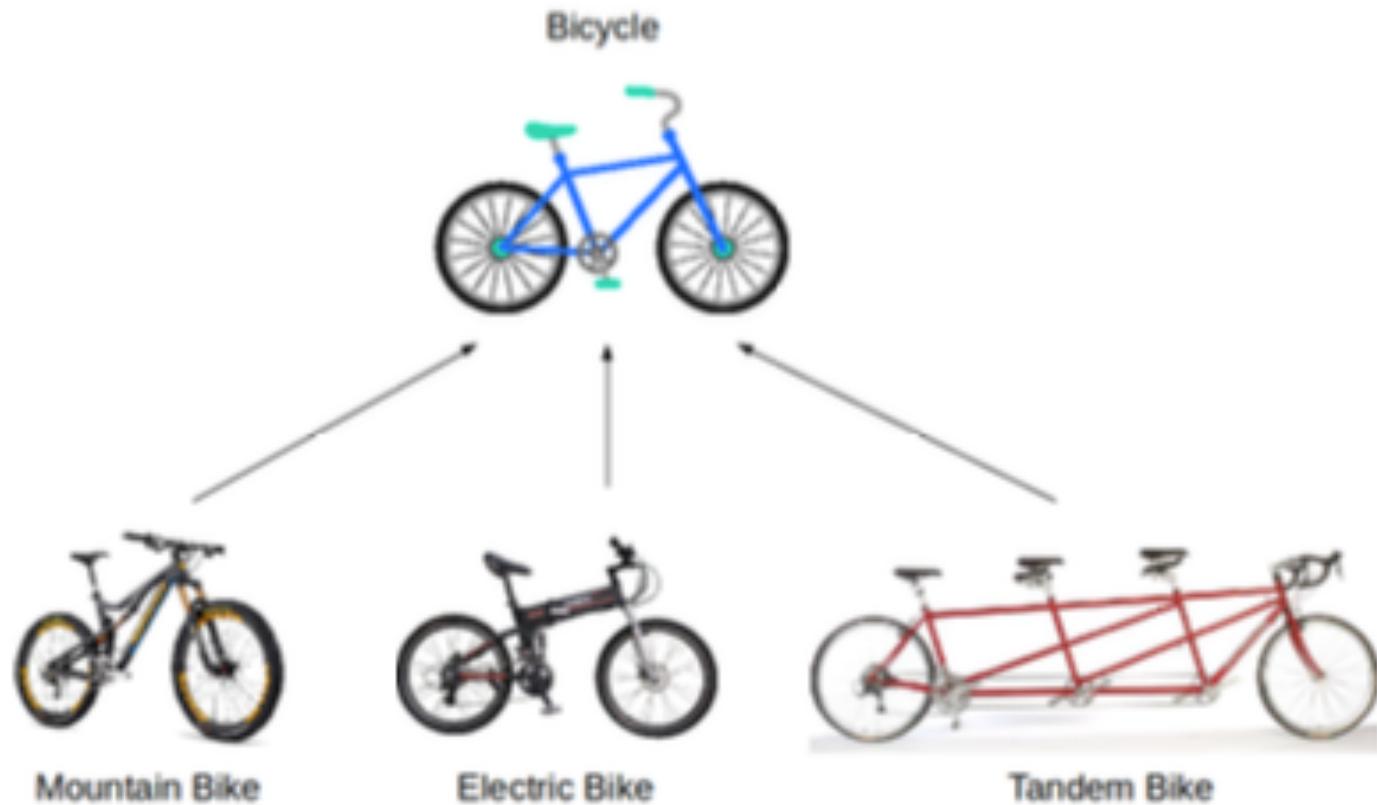
Herança

- Herança é o mecanismo através do qual elementos mais específicos incorporam a estrutura e comportamento de elementos mais gerais
 - Super-Classe (classe mãe ou classe base) - Elemento mais geral utilizado para evitar as redundâncias de descrição.
 - Sub-Classe (classe filha ou classe) Elemento mais específico utilizado para descrever as particularidades das classes.
- A sub-classe pode ser utilizada onde a super-classe é mas não o contrário.
- Serve para reutilizar campos e métodos já criados, descrevendo as diferenças entre classes existentes e novas classes
- Um objeto de uma subclasse também é um objeto de uma superclasse.

Herança

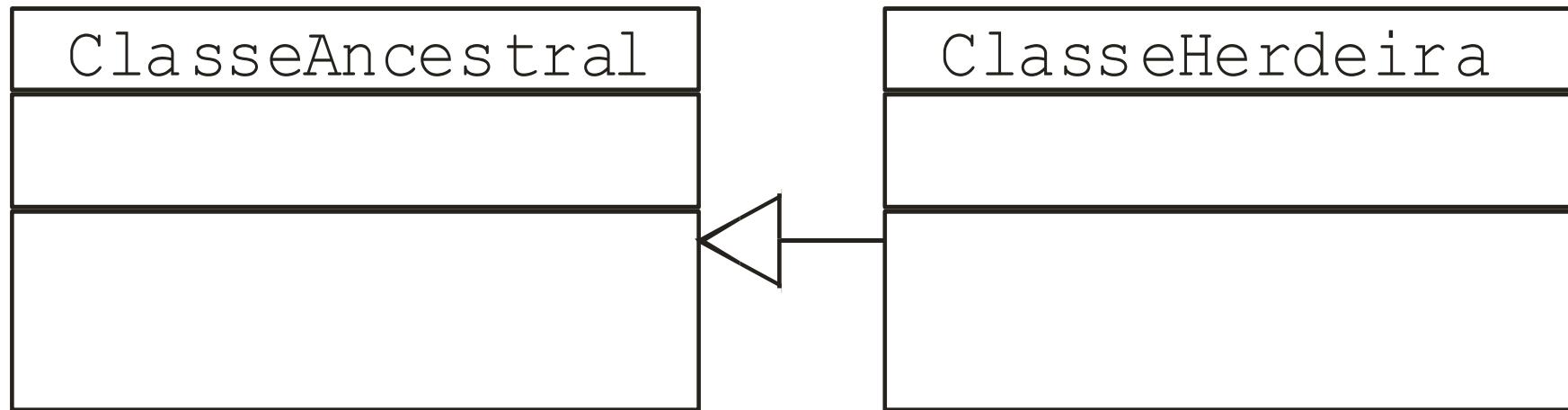
- Atributos, operações e relacionamentos comuns devem ser representados no mais alto nível da hierarquia
- O que é herdado:
 - As sub-classes criadas herdam o estado (atributos e relacionamentos) e o comportamento (operações) da(s) super-classe(s) associada(s) (herança simples ou múltipla)
 - As sub-classes podem incluir, alterar ou suprimir atributos, operações e relacionamentos da super-classe
 - Os métodos construtores não são herdados, mas são acessíveis às subclasses

Herança



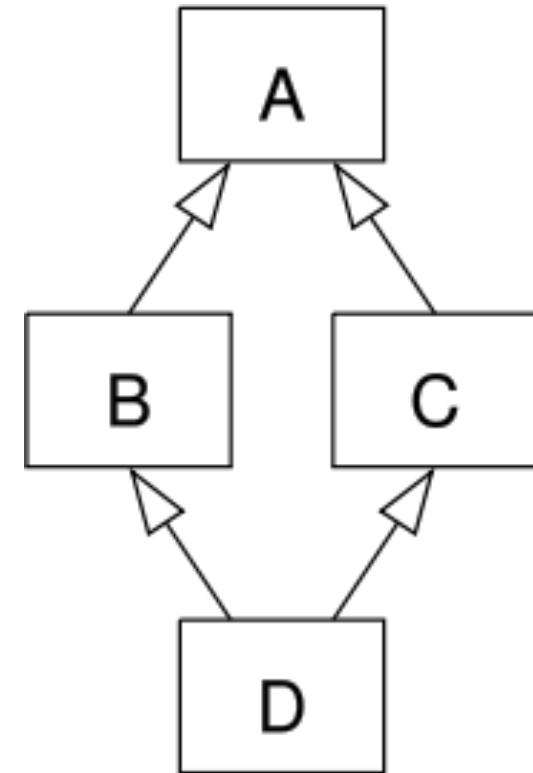
Herança: Representação

- **ClasseHerdeira** herda todos os campos e métodos da **ClasseAncestral**
- Classe herdeira só pode acessar diretamente campos e métodos públicos da classe ancestral!
- Importância de métodos setXXX e getXXX para acessar e modificar valores de campos



Herança Múltipla

- **Herança Múltipla:** classe herdeira herda de duas ou mais classes ancestrais: não existe em Java mas é possível em C++.
- Uma classe ancestral pode ter várias classes descendentes
- Uma classe herdeira pode ter, por sua vez, outras classes herdeiras.
- Não é considerada uma boa solução de projeto.

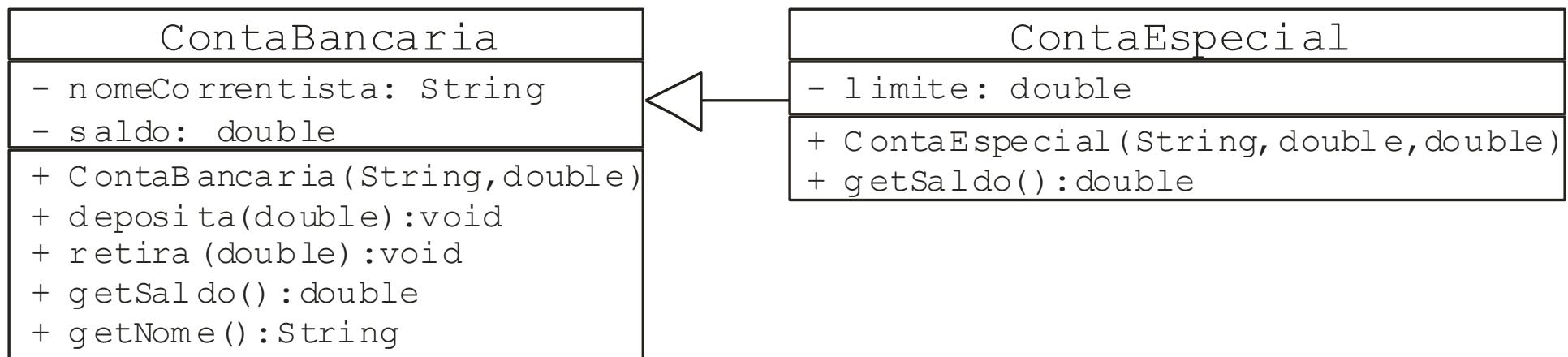


Herança: Exemplos

- ContaEspecial é um tipo de ContaBancaria
- ContaPoupanca é um tipo de ContaBancaria
- Gerente é um tipo de Funcionario
- LivroParaEmprestimo é um tipo de ItemDoAcervo

Exemplo 1

- ContaEspecial herdará todos os métodos de ContaBancaria menos o construtor e mais os definidos para ContaEspecial
- Conta especial simulada como tendo saldo adicional (limite)
- Problema: método getSaldo() de ContaBancaria não serviria para ContaEspecial!
- Solução: sobrescrita do método.



Exemplo 1: ContaBancaria

```
class ContaBancaria
{
    private:
        std::string nomeCorrentista;
        double saldo;
    public:
        ContaBancaria(std::string n, double s);
        void deposita(double quantia);
        void retira(double quantia);
        double getSaldo();
        std::string getNome();
};
```

Exemplo 1: ContaBancaria

```
#include "ContaBancaria.h"

ContaBancaria::ContaBancaria(std::string n,double s)
{
    nomeCorrentista = n;
    saldo = s;
}

void ContaBancaria::deposita(double quantia)
{
    saldo = saldo + quantia;
}

void ContaBancaria::retira(double quantia)
{
    if (quantia < saldo)
        saldo = saldo - quantia;
}

double ContaBancaria::getSaldo()
{
    return saldo;
}

std::string ContaBancaria::getNome()
{
    return nomeCorrentista;
}
```

Exemplo 1: ContaEspecial

```
class ContaEspecial: public ContaBancaria
{
    private:
        double limite;
    public:
        ContaEspecial(std::string n, double s, double l);
        double getSaldo();
};
```

Indica qual classe será usada como base (ancestral) para herança

Chamada do **construtor** da classe ancestral com os argumentos

Chamada do método **getSaldo** da classe ancestral

Exemplo 1: ContaEspecial

```
ContaEspecial::ContaEspecial(std::string n, double s, double l) :  
    ContaBancaria(n, s)  
{  
    limite = l;  
    deposita(limite);  
}  
  
double ContaEspecial::getSaldo()  
{  
    return ContaBancaria::getSaldo() - limite; // saldo verdadeiro  
}
```

Chamada do **construtor** da classe ancestral com os argumentos

Chamada do método **getSaldo** da classe ancestral

Herança

- A subclasse ContaEspecial possui os atributos e métodos da superclasse ContaBancaria, mas o adicional da subclasse só implementa o que for necessário na mesma
 - Isso torna o código da subclasse mais fácil de entender
 - tem um atributo a mais que a superclasse
 - limite

Herança: Construtores

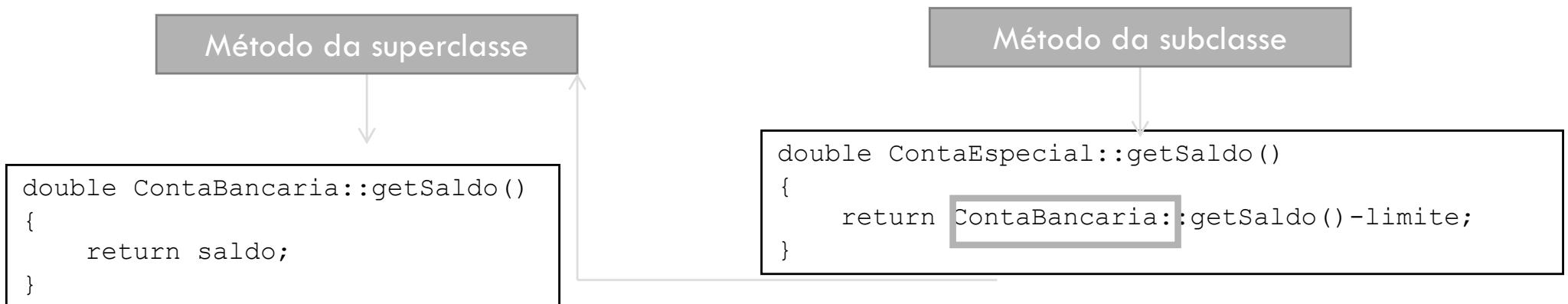
- Os métodos construtores não são herdados, mas são acessíveis às subclasses
 - Chamada de construtores da superclasse
 - A primeira tarefa de um construtor de uma subclasse é chamar o construtor da superclasse, explícita ou implicitamente, para assegurar que as variáveis de instância herdadas serão inicializadas corretamente
 - Se o construtor da subclasse não chama o construtor da superclasse explicitamente, então o compilador gera uma instrução que chama o construtor default ou o construtor sem argumento da superclasse. Se não houver tal construtor na superclasse ocorre um erro de compilação

Herança: Sobrescrita de métodos

- Sobrescrita (override) está diretamente relacionada à herança
- Permite especializar os métodos herdados das superclasses, alterando o seu comportamento nas subclasses por um mais específico.
 - Cria-se um novo método na classe filha contendo a mesma assinatura e mesmo tipo de retorno do método sobrescrito
 - O método deve possuir a mesma assinatura
 - nome
 - quantidade de parâmetros
 - tipo de parâmetros
- A declaração de um método como virtual indica que ele pode ter implementações diferentes em classes derivadas
- Para definir um método virtual em C++ usa-se a palavra-chave `virtual` na declaração do método na classe base

Herança: Sobrescrita de métodos

- Como se sabe que método invocar?
 - Primeiramente busca-se um método com o nome indicado na classe atual
 - Caso o método exista na subclasse, este método será executado
 - Caso contrário, busca-se e se o método na superclasse



Sobrescrita x Sobrecarga

- Sobrescrita
 - Mesmo nome
 - Mesma assinatura
 - Requer herança
- Sobrecarga
 - Mesmo nome
 - Assinaturas diferentes (ordem ou tipo dos parâmetros – não inclui o retorno)
 - Mesma classe ou na hierarquia

Exemplo 1: ContaEspecial

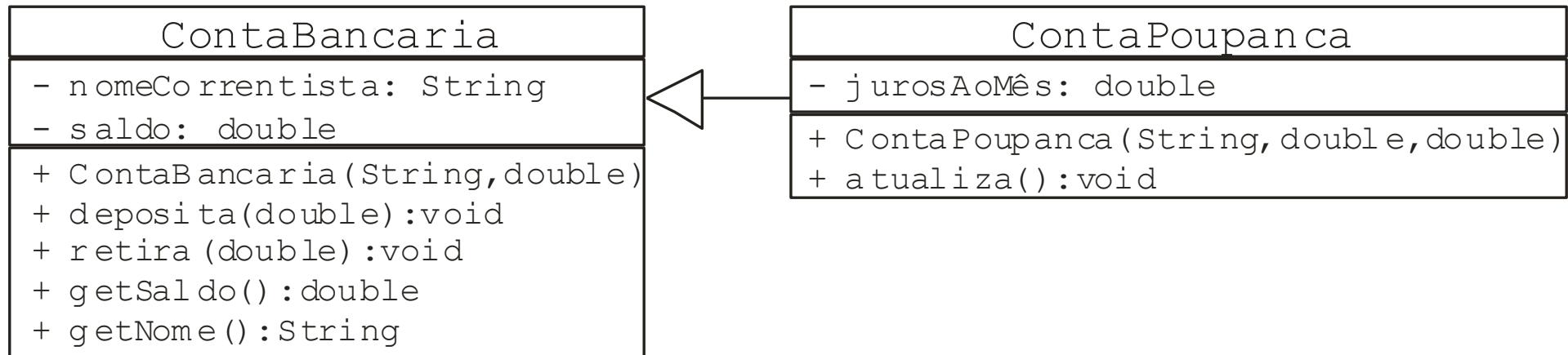
80
-1000

```
int main()
{
    ContaBancaria minha("Eu", 200);
    minha.retira(120);
    minha.retira(100);
    std::cout << minha.getSaldo() << std::endl;; // 80
    ContaEspecial sua("Você", 10000, 2000);
    sua.retira(5000);
    sua.retira(6000);
    std::cout << sua.getSaldo() << std::endl;; // -1000

}
```

Exemplo 2: ContaPoupanca

- Herda praticamente tudo de ContaBancaria
- Não sobrepõe nenhum método
- Método atualiza calcula juros e atualiza saldo (deve ser chamado todo mês)



Exemplo 2: ContaPoupanca

```
class ContaPoupanca:public ContaBancaria
{
    private:
        double juros;
    public:
        ContaPoupanca(std::string n,double s,double j);
        void atualiza();
};
```

Indica qual classe será usada como base
(ancestral) para herança

```
ContaPoupanca::ContaPoupanca (std::string n,double s,double j):
    ContaBancaria(n,s)
```

Chamada do **construtor** da classe ancestral com
os argumentos

```
void ContaPoupanca::atualiza()
{
    deposita(getSaldo()*juros);
}
```

Chamada do método **getSaldo** da
classe ancestral

Chamada do método **deposita** da classe
ancestral com o argumento

Exemplo 2: Main

```
int main()
{
    ContaBancaria minha("Eu", 200);
    minha.retira(120);
    minha.retira(100);
    std::cout << minha.getSaldo() << std::endl;; // 80
    ContaEspecial sua("Você", 10000, 2000);
    sua.retira(5000);
    sua.retira(6000);
    std::cout << sua.getSaldo() << std::endl;; // -1000
    ContaPoupanca outra("Outra", 10000, 0.1);
    outra.atualiza();
    std::cout << outra.getSaldo() << std::endl;; // -11000
}
```

80
-1000
11000

Outro exemplo

```
#include <iostream>
class Animal {
public:
    virtual void fazerSom() {
        std::cout << "Animal fazendo um som." << std::endl;
    }
};

class Cachorro : public Animal {
public:
    void fazerSom() override {
        std::cout << "Cachorro latindo." << std::endl;
    }
};

class Gato : public Animal {
public:
    void fazerSom() override {
        std::cout << "Gato miando." << std::endl;
    }
};

int main() {
    Cachorro cachorro;
    Gato gato;
    cachorro.fazerSom(); // Chama o método fazerSom() da classe Cachorro
    gato.fazerSom(); // Chama o método fazerSom() da classe Gato
    return 0;
}
```

Cachorro latindo.
Gato miando.

Herança Múltipla

```
#include <iostream>
class ClasseBase1 {
public:
    virtual void metodo() {
        std::cout << "Chamando método da ClasseBase1" << std::endl;
    }
};

class ClasseBase2 {
public:
    virtual void metodo() {
        std::cout << "Chamando método da ClasseBase2" << std::endl;
    }
};

// Classe derivada com herança múltipla
class ClasseDerivada : public ClasseBase1, public ClasseBase2 {
public:
    void metodo() override {
        std::cout << "Chamando método da ClasseDerivada" << std::endl;
    }
};

int main() {
    ClasseDerivada instancia;
    // A chamada a 'metodo' na ClasseDerivada irá sobrescrever
    // as implementações de ClasseBase1 e ClasseBase2
    instancia.metodo(); // Método da ClasseDerivada
    // Para chamar explicitamente os métodos de ClasseBase1 ou ClasseBase2
    instancia.ClasseBase1::metodo(); // Método da ClasseBase1
    instancia.ClasseBase2::metodo(); // Método da ClasseBase2
    return 0;
}
```

Chamando método da ClasseDerivada
Chamando método da ClasseBase1
Chamando método da ClasseBase2

A palavra final

- A palavra final também pode ser usada, em diferentes circunstâncias:
 - Quando usada na definição de uma classe, significa que a classe não vai admitir herança.

```
class ClasseBase final {  
    // ...  
};  
  
class ClasseDerivada : public ClasseBase {  
    // Isso resultaria em erro de compilação,  
    // pois não é permitido derivar de uma classe final  
};
```

A palavra final

- A palavra final também pode ser usada, em diferentes circunstâncias:
 - Quando usada na definição de um método, significa que o método não poderá ser sobreescrito.

```
class ClasseBase {  
public:  
    virtual void metodoVirtual() {  
        // Implementação padrão  
    }  
};  
  
class ClasseDerivada : public ClasseBase {  
public:  
    void metodoVirtual() final {  
        // Implementação na classe derivada  
    }  
};  
  
class OutraClasse : public ClasseDerivada {  
public:  
    // Isso resultaria em erro de compilação,  
    // pois não é permitido sobreescrivar um método final  
    void metodoVirtual() override {}  
};
```

- Na próxima aula
 - Relacionamentos