

Curso de Extensão - INF1900

Programação em C++



Módulo 2

Programação Orientada a Objetos em C++

Aulas 3 & 4

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.

Relacionamentos

Sumário

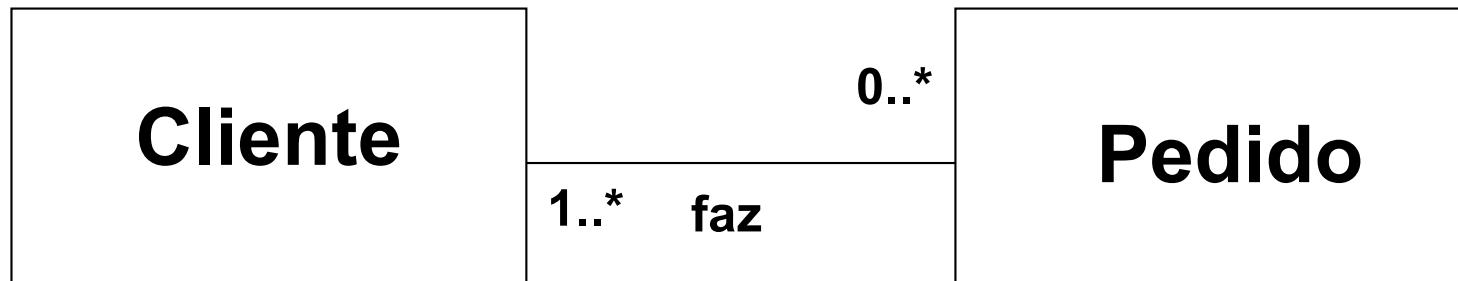
- Relacionamentos
- Herança (Revisão)
- Ligação Dinâmica
- Polimorfismo

Associação, Agregação e Composição: Conceitos

- Representam relações entre objetos
 - Associação
 - Agregação
 - Composição
- Nem sempre a distinção é clara!

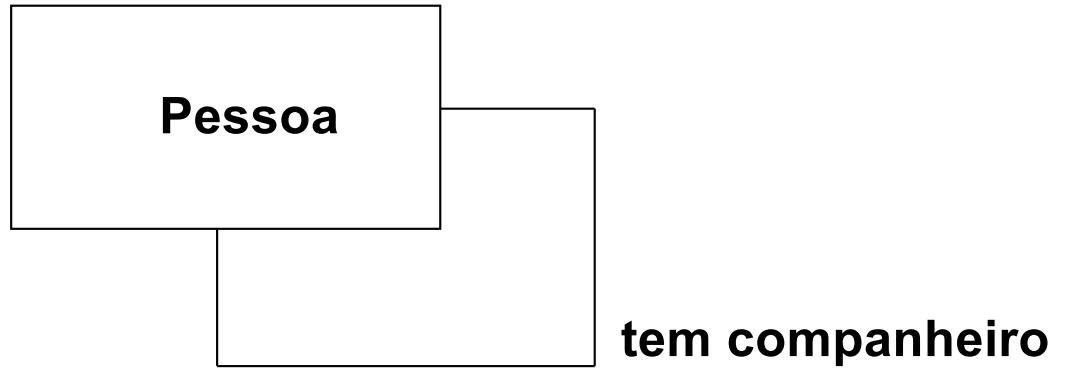
Associação

- Associação existente entre duas entidades
 - Cliente possui Pedidos. Um pedido é referente a um cliente.



Associação - tipos

- associação unária

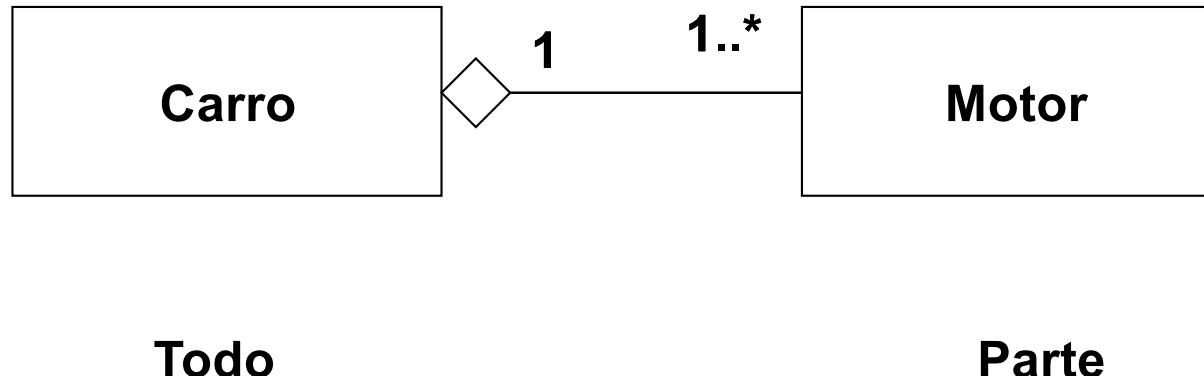


- associação binária



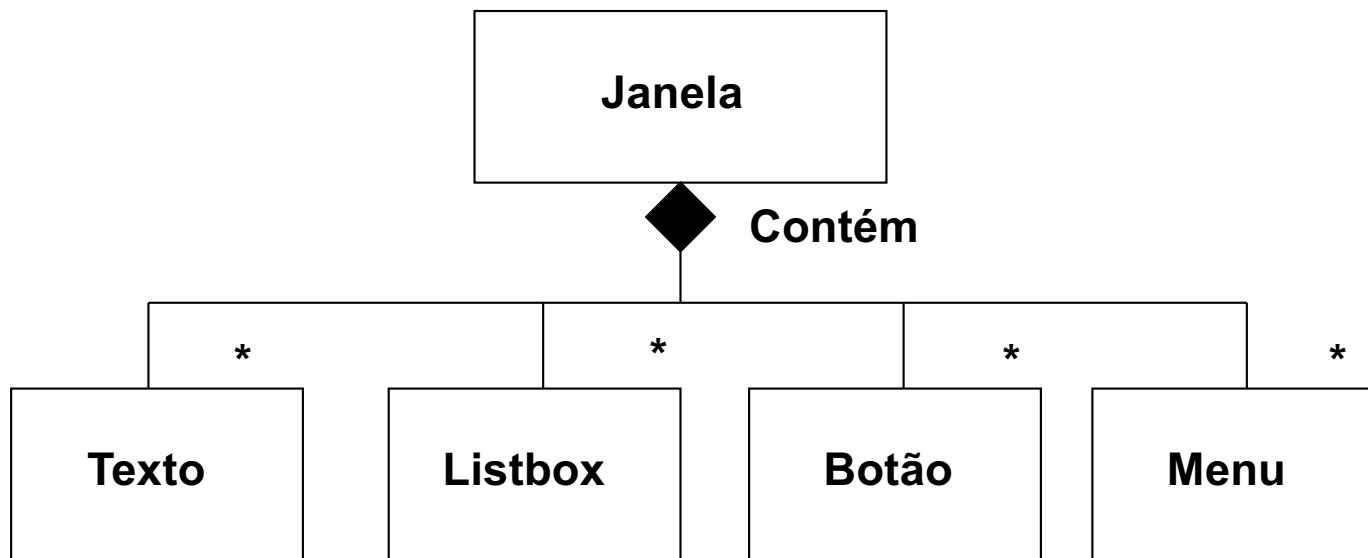
Agregação

- Considerar algo maior pensando na relação todo-parte. Indica que um objeto parte é um atributo do objeto todo.
 - Um Carro C possui um Motor M. Se o carro for removido do sistema, o motor que estava sendo desenvolvido pode ser reaproveitado em outro carro.



Composição

- Também baseado na relação todo-parte. Neste caso, se o objeto maior for removido, as suas partes filhas serão removidas também.
 - Imagine o caso de um Carro C que possui uma Placa P registrada. Se o carro deixa de existir, a placa não tem mais utilidade dentro do sistema.

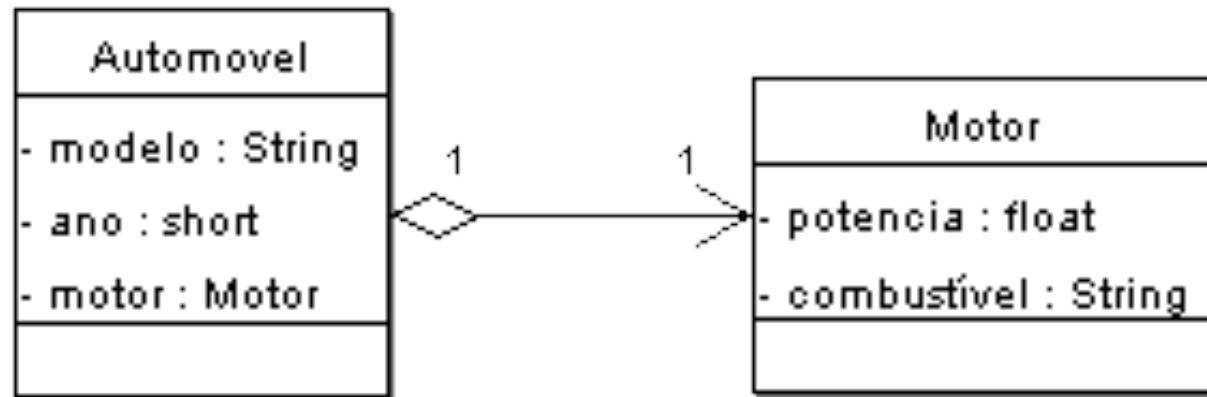


Associação, Agregação e Composição: Conceitos

- **Associação:** é uma relação fraca entre dois objetos, o que significa que eles podem existir independentemente um do outro.
- **Agregação:** é uma relação mais forte do que a associação, mas ainda permite que os objetos envolvidos existam independentemente.
- **Composição:** é a relação mais forte entre objetos e implica uma propriedade de propriedade. Um objeto "todo" possui diretamente os objetos "partes" e é responsável por criá-los e destruí-los.
- Além disso:
 - Diferenças na implementação:
 - Multiplicidade de associações: 1 para 1, 1 para *, * para * (* = muitos).
 - Implementação diferente para * fixo e variável.

Exemplo 1

- Um automóvel deve ter um motor instalado.
 - Uma instância de motor só pode ser associada a uma instância de automóvel a qualquer momento, e um automóvel só pode possuir um motor a cada instante: **relação 1 para 1**.



Exemplo 1: Classe Motor

```
#include <iostream>

class Motor
{
    private:
        float potencia;
        std::string combustivel;
    public:
        Motor(float pot, std::string comb);
        void imprimeDados();
};

Motor::Motor(float pot, std::string comb) {
    potencia = pot;
    combustivel = comb;
}

void Motor::imprimeDados() {
    std::cout << "Motor com potência " << potencia << "movido a " <<
    combustivel;
}
```

Exemplo 1: Classe Automovel

```
class Automovel
{
private:
    std::string modelo;
    int ano;
    Motor *motor;
public:
    Automovel(std::string mod, short a, Motor *mot);
    void imprimeDados();
};

Automovel::Automovel(std::string mod, short a, Motor *mot:
modelo(mod), ano(a), motor(mot) {}

void Automovel::imprimeDados()
{
    std::cout << "Modelo:" << modelo << "Ano:" << ano ;
    motor->imprimeDados();
}
```

Exemplo 1: Teste de Automóvel

```
int main(void)
{
    Motor *motFusca = new Motor(47,"gasolina");
    Automovel *fusca66 = new Automovel("Fusca",1966,motFusca);
    fusca66->imprimeDados();

    Automovel beetle2002("New Beetle", 2002,new Motor(150,"gasolina"));
    beetle2002.imprimeDados();
    return 0;
}
```

Modelo: Fusca, ano: 1966

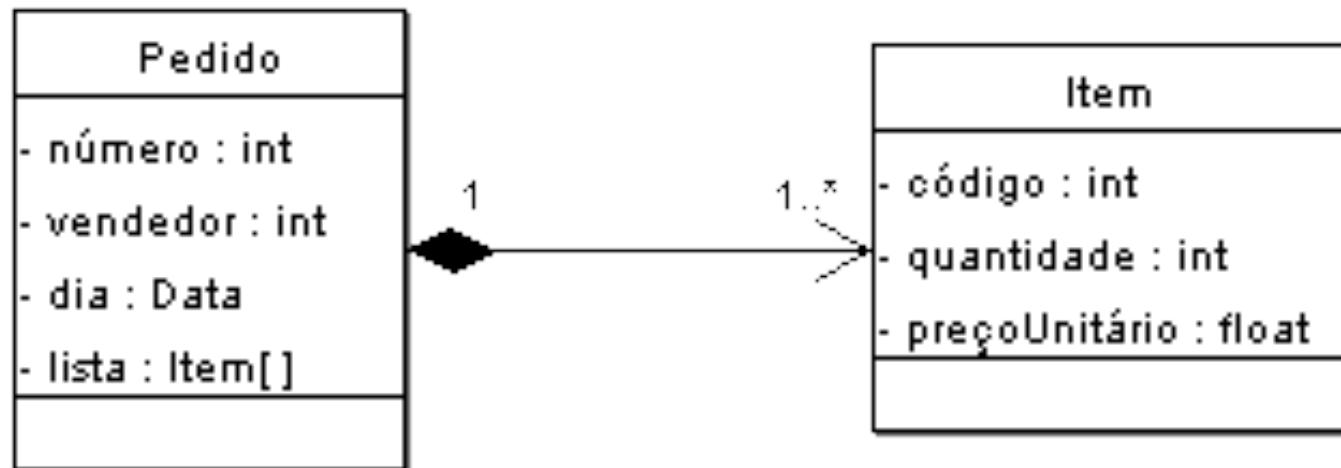
Motor com potência 47 cavalos, movido a gasolina

Modelo: New Beetle, ano: 2002

Motor com potência 150 cavalos, movido a gasolina

Exemplo 2

- Um pedido de compras deve ter vários itens.
 - Não podemos ter **pedidos** vazios e um **item** deve pertencer a um único pedido: **relação 1 para 1..***.



Exemplo 2: Relação 1 para 1..*

- Como representar esta relação ?

```
class Pedido
{
    private:
        int vendedor;
        Data dia;
        Item lista1;
        Item lista2;
        Item lista3;
        Item lista4;
        Item lista5;
        // ...
}
```

- Como saber que instâncias estão em uso ?
- E se precisar de mais instâncias de Item do que as declaradas ?

A classe Vector

- Pode ser usada para armazenar 0 a N instâncias de qualquer classe usando somente uma referência.
- Contém métodos para inserir, remover e recuperar as instâncias armazenadas.
- Elementos são indexados de 0 a N-1 (onde N é o número de elementos no Vector)

Exemplo 2: Classe Data

```
#ifndef DATA_H
#define DATA_H

#include <iostream>

class Data {
private:
    int dia,mes;
    int ano;
public:
    Data(int d,int m,int a);
    void imprime();
};

#endif
```

Exemplo 2: Classe Data

```
#include "Data.h"

Data::Data(int d,int m,int a) {
    dia = d;  mes = m;  ano = a;
}

void Data::imprime() {
    std::cout << "Data: " << dia<< "/" << mes << "/" << ano << std::endl;
}
```

Exemplo 2: Classe Item

```
#ifndef ITEM_H
#define ITEM_H

class Item {
private:
    int codigo;
    int quantidade;
    float precoUnitario;

public:
    Item(int cod,int quant,float preco);
    void imprime();
    float custoTotal();
};

#endif
```

Exemplo 2: Classe Item

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

Item::Item(int cod, int quant, float preco) {
    codigo = cod; quantidade = quant; precoUnitario = preco;
}

void Item::imprime() {
    std::cout << "Item: codigo " << codigo << " " <<
        quantidade << " unidades a " <<
        precoUnitario << " cada." << std::endl;
}

float Item::custoTotal() {
    return quantidade*precoUnitario;
}
```

Exemplo 2: Classe Pedido

```
#ifndef PEDIDO_H
#define PEDIDO_H

#include <iostream>
#include <vector>
#include "Item.h"
#include "Data.h"

class Pedido {
private:
    int numero;
    int vendedor;
    Data *dia;
    std::vector <Item *> lista;
public:
    Pedido(int n,int v,Data *d);
    void adicionaItem(int cod,int quant,float preco);
    float calculaTotal();
    void imprime();
};

#endif
```

Exemplo 2: Classe Pedido

```
#include "Pedido.h"
Pedido::Pedido(int n,int v,Data *d):numero(n),vendedor(v),dia(d) {}
void Pedido::adicionaItem(int cod,int quant,float preco) {
    lista.push_back(new Item(cod,quant,preco));
}

float Pedido::calculaTotal() {
    float total = 0;
    for(int qual=0;qual<lista.size();qual++) {
        Item *umItem = lista.at(qual);
        total = total + umItem->custoTotal();
    }
    return total;
}

void Pedido::imprime() {
    std::cout<< "Pedido #" << numero << " do vendedor " << vendedor << std::endl;
    dia->imprime();
    std::cout<< "Itens:" << std::endl;
    for(int qual=0;qual<lista.size();qual++) {
        Item *umItem = lista.at(qual);
        std::cout<< " * ";
        umItem->imprime();
    }
    std::cout << "Total do pedido: " << calculaTotal() << std::endl;
}
```

Exemplo 2: Teste do Pedido

```
#include <iostream>
#include "Data.h"
#include "Item.h"
#include "Pedido.h"

int main()
{
    Data *hoje = new Data(19, 8, 2023);
    Pedido *p = new Pedido(1,3,hoje);

    p->adicionaItem(1215,10, 9.45);
    p->adicionaItem(1217, 1,21.00);
    p->adicionaItem(1223, 1,22.05);
    p->adicionaItem(1249, 3,50.95);

    p->imprime();
    return 0;
}
```

Pedido #1 do vendedor 3
Data: 19/8/2023
Itens:

* Item: codigo 1215 10 unidades a 9.45 cada.

* Item: codigo 1217 1 unidades a 21 cada.

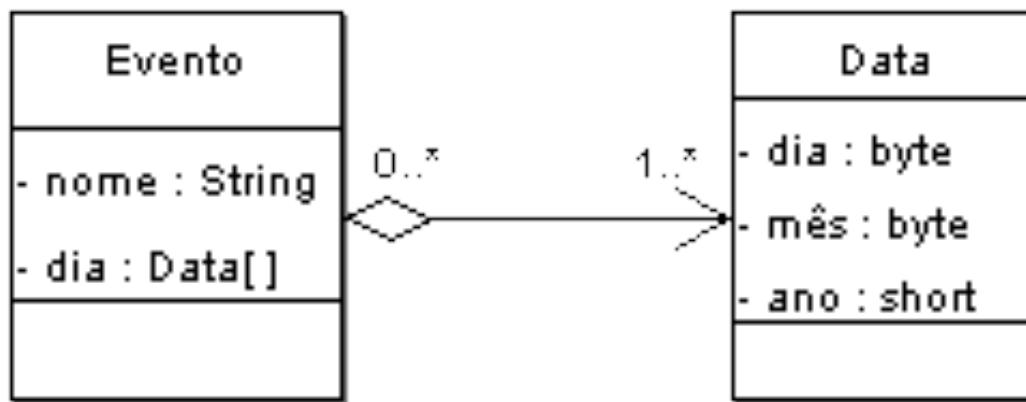
* Item: codigo 1223 1 unidades a 22.05 cada.

* Item: codigo 1249 3 unidades a 50.95 cada.

Total do pedido: 290.4

Exemplo 3

- Um evento ocorre em uma determinada data.
 - Uma instância de data pode ser usada por várias instâncias de evento ou não usada por nenhuma. Um evento pode ter mais de uma data:**relação 0..* para 1..*.**



Exemplo 3: Classe Evento

```
#ifndef EVENTO_H
#define EVENTO_H

#include <iostream>
#include <vector>
#include "Data.h"

class Evento {
private:
    std::string nome;
    std::vector <Data *> dia;
public:
    Evento(std::string n, Data *umDia);
    void marcaDiaAdicional(Data *d);
    void imprime();
};

#endif
```

Exemplo 3: Classe Evento

```
#include "Evento.h"

Evento::Evento(std::string n, Data *umDia) {
    nome = n;
    dia.push_back(umDia);
}

void Evento::marcaDiaAdicional(Data *d) {
    dia.push_back(d);
}

void Evento::imprime() {
    std::cout << "Evento: " << nome << " ocorrerá nos
dias:" << std::endl;
    for (int d = 0; d < dia.size(); d++) {
        std::cout << " * ";
        Data *umaData = dia.at(d);
        umaData->imprime();
    }
}
```

Exemplo 3: Teste do Evento

```
#include <iostream>
#include "Data.h"
#include "Evento.h"

int main() {
    Evento *encontro = new Evento("Encontro de Pessoas que
Realmente Amam Programar", new Data(1, 4, 2010));
    encontro->marcaDiaAdicional(new Data(2, 4, 2023));
    encontro->marcaDiaAdicional(new Data(3, 4, 2023));
    encontro->marcaDiaAdicional(new Data(4, 4, 2023));
    encontro->imprime();
}
```

Evento: Encontro de Pessoas que Realmente Amam Programar ocorrerá nos dias:

- * Data: 1/4/2010
- * Data: 2/4/2023
- * Data: 3/4/2023
- * Data: 4/4/2023

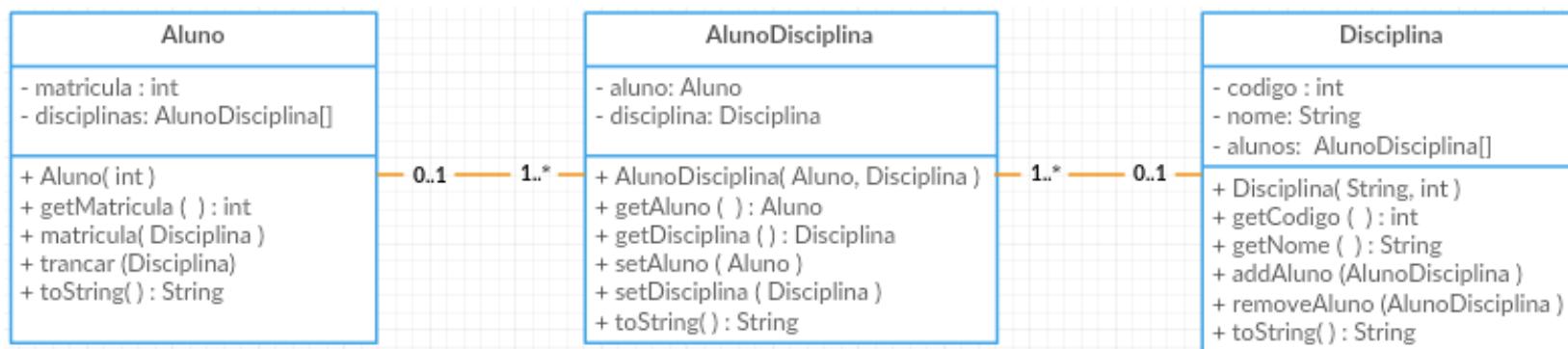
Relacionamentos Bidirecionais

Exemplo Aluno-Disciplina

- Usando uma classe Associativa
 - Quando a associação entre duas classes possui multiplicidade muitos (*) em ambas as extremidades, uma forma de se manter esta estrutura ocorre por meio de uma **classe associativa**.
 - Esta classe é necessária para armazenar atributos transmitidos pela associação de ambas as classes, podendo a mesma, inclusive, possuir atributos próprios.

Exemplo Aluno-Disciplina

- Usando uma classe Associativa
 - 1 aluno pode cursar várias disciplinas
 - 1 disciplina pode ter vários alunos



Aluno.h

```
#ifndef ALUNO_H
#define ALUNO_H

#include <iostream>
#include <vector>
#include "AlunoDisciplina.h"
#include "Disciplina.h"

class Aluno {
private:
    int matricula;
    std::vector<class AlunoDisciplina> disciplinas;
public:
    Aluno(int m);
    int getMatricula();
    void matricular(class Disciplina& d);
    void trancar(class Disciplina& d);
    std::string toString();
};

#endif
```

Disciplina.h

```
#ifndef DISCIPLINA_H
#define DISCIPLINA_H

#include <iostream>
#include <vector>
#include "AlunoDisciplina.h"

class Disciplina {
private:
    std::string nome;
    int codigo;
    std::vector<class AlunoDisciplina> alunos;

public:
    Disciplina(std::string n, int c);
    std::string getNome();
    int getCodigo();
    void addAluno(AlunoDisciplina& a);
    void removeAluno(AlunoDisciplina& a);
    std::string toString();
};

#endif
```

AlunoDisciplina.h

```
#ifndef ALUNO_DISCIPLINA_H
#define ALUNO_DISCIPLINA_H

#include <iostream>
#include <vector>
#include "Aluno.h"
#include "Disciplina.h"

class AlunoDisciplina {
private:
    class Aluno* aluno;
    class Disciplina* disciplina;

public:
    AlunoDisciplina(Aluno* a, class Disciplina* d);
    Aluno* getAluno();
    Disciplina* getDisciplina();
    void setAluno(class Aluno* a);
    void setDisciplina(class Disciplina* d);
};

#endif
```

Aluno.cpp

```
#include "Aluno.h"
Aluno::Aluno(int m) : matricula(m) {}
int Aluno::getMatricula() {
    return matricula;
}
void Aluno::matricular(Disciplina& d) {
    AlunoDisciplina ad(this, &d);
    disciplinas.push_back(ad);
    d.addAluno(ad);
}
void Aluno::trancar(Disciplina& d) {
    for (int i = 0; i < disciplinas.size(); ++i) {
        if (disciplinas[i].getDisciplina() == &d) {
            d.removeAluno(disciplinas[i]);
            disciplinas.erase(disciplinas.begin() + i);
            break;
        }
    }
}
std::string Aluno::toString() {
    std::string out = "Aluno com matrícula # " + std::to_string(getMatricula()) + " está matriculado\nnas disciplinas:\n";
    for (int i = 0; i < disciplinas.size(); i++) {
        out += " * " + disciplinas[i].getDisciplina()->getNome() + "\n";
    }
    return out;
}
```

Disciplina.cpp

```
#include "Disciplina.h"
class AlunoDisciplina;
Disciplina::Disciplina(std::string n, int c) : nome(n), codigo(c) {}
std::string Disciplina::getNome() {
    return nome;
}
int Disciplina::getCodigo() {
    return codigo;
}
void Disciplina::addAluno(class AlunoDisciplina& a) {
    alunos.push_back(a);
}
void Disciplina::removeAluno(class AlunoDisciplina& a) {
    for (int i = 0; i < alunos.size(); ++i) {
        if (alunos[i].getAluno() == a.getAluno()) {
            alunos.erase(alunos.begin() + i);
            break;
        }
    }
}
std::string Disciplina::toString() {
    std::string out = "Alunos matriculados na disciplina código " + std::to_string(getCodigo()) + " "
+ getNome() + ":\n";
    for (int i = 0; i < alunos.size(); i++) {
        out += " * " + std::to_string(alunos[i].getAluno()->getMatricula()) + "\n";
    }
    return out;
}
```

AlunoDisciplina.cpp

```
#include "AlunoDisciplina.h"

AlunoDisciplina::AlunoDisciplina(class Aluno* a, class Disciplina*
d) : aluno(a), disciplina(d) {}

Aluno* AlunoDisciplina::getAluno() {
    return aluno;
}

Disciplina* AlunoDisciplina::getDisciplina() {
    return disciplina;
}

void AlunoDisciplina::setAluno(Aluno* a) {
    aluno = a;
}

void AlunoDisciplina::setDisciplina(Disciplina* d) {
    disciplina = d;
}
```

Teste.cpp

```
#include <iostream>
#include <vector>
#include "Aluno.h"
#include "AlunoDisciplina.h"
#include "Disciplina.h"

int main() {
    Disciplina pp("Programação em Prolog", 11001);
    Disciplina pl("Programação em Lisp", 11002);
    Disciplina ia("Inteligência Artificial", 11201);
    Disciplina ln("Lógica Nebulosa", 11205);
    Disciplina ag("Algoritmos Genéticos", 11760);

    Aluno m(34030001);
    m.matricular(pp);
    m.matricular(ia);
    m.matricular(ln);
    std::cout << m.toString() << std::endl;

    Aluno n(34030029);
    n.matricular(pl);
    n.matricular(ln);
    n.matricular(ag);
    std::cout << n.toString() << std::endl;
```

Aluno com matrícula # 34030001 está matriculado nas disciplinas:
* Programação em Prolog
* Inteligência Artificial
* Lógica Nebulosa

Aluno com matrícula # 34030029 está matriculado nas disciplinas:
* Programação em Lisp
* Lógica Nebulosa
* Algoritmos Genéticos

Aluno com matrícula # 34030088 está matriculado nas disciplinas:
* Programação em Prolog
* Inteligência Artificial
* Algoritmos Genéticos

Teste.cpp

```
Aluno o(34030088);
o.matricular(pp);
o.matricular(ia);
o.matricular(ag);
std::cout << o.toString() << std::endl;
std::cout << pp.toString() << std::endl;
std::cout << pl.toString() << std::endl;
std::cout << ia.toString() << std::endl;
std::cout << ln.toString() << std::endl;
std::cout << ag.toString() << std::endl;
std::cout << "***** Depois do trancamento *****\n";
o.trancar(pp);
std::cout << o.toString() << std::endl;
std::cout << pp.toString() << std::endl;
return 0;
}
```

Teste.cpp

Alunos matriculados na disciplina código 11001 Programação em Prolog:

- * 34030001
- * 34030088

Alunos matriculados na disciplina código 11002 Programação em Lisp:

- * 34030029

Alunos matriculados na disciplina código 11201 Inteligência Artificial:

- * 34030001
- * 34030088

Alunos matriculados na disciplina código 11205 Lógica Nebulosa:

- * 34030001
- * 34030029

Alunos matriculados na disciplina código 11760 Algoritmos Genéticos:

- * 34030029
- * 34030088

***** Depois do trancamento *****

Aluno com matrícula # 34030088 está matriculado nas disciplinas:

- * Inteligência Artificial
- * Algoritmos Genéticos

Alunos matriculados na disciplina código 11001 Programação em Prolog:

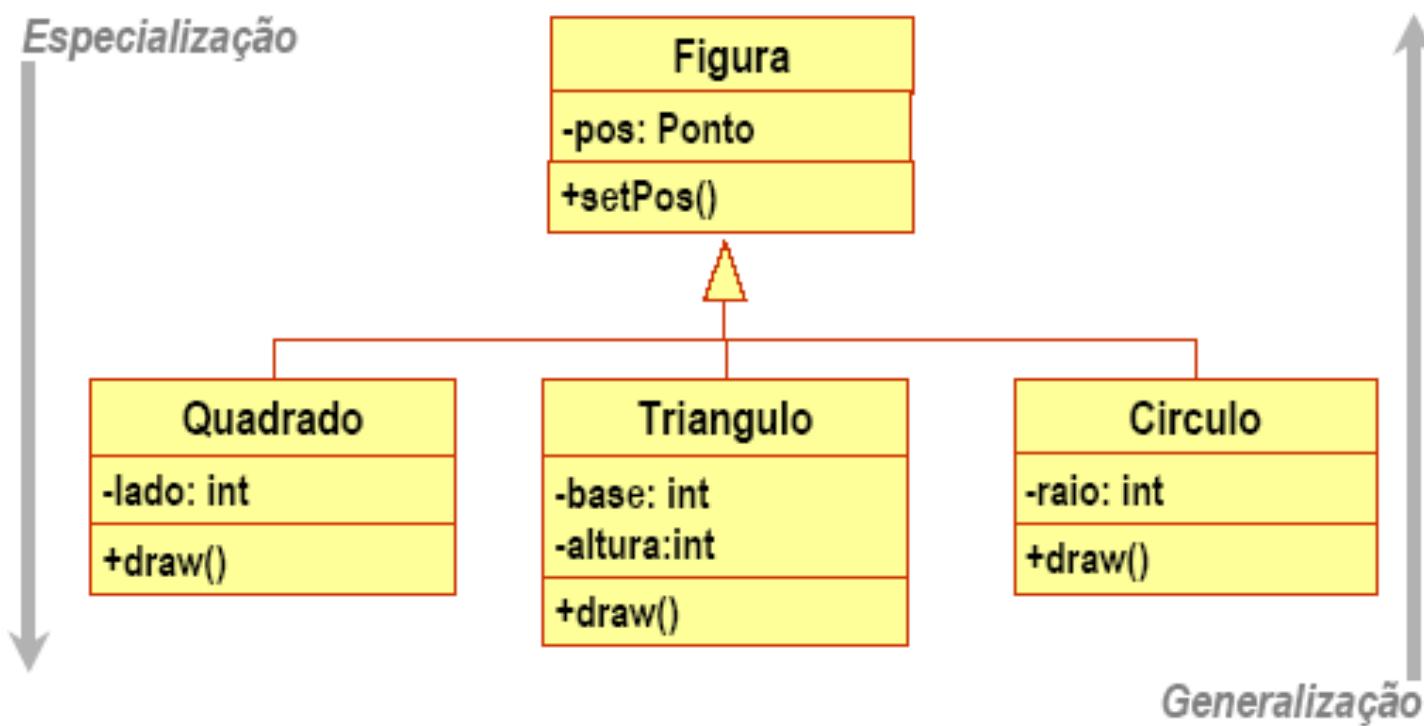
- * 34030001

Herança (Revisão)

- Forma de reutilização de código
 - Propriedades e métodos são herdados, evitando duplicidade em classes similares
- Forma de generalização/especialização de tipos
 - Várias classes podem ser generalizadas em uma classe que concentre todas as características comuns a todas elas
 - Uma classe pode ser especializada em várias outras
- Pelo menos duas classes são requeridas para que haja herança!

Herança (Revisão)

- Representação de Herança



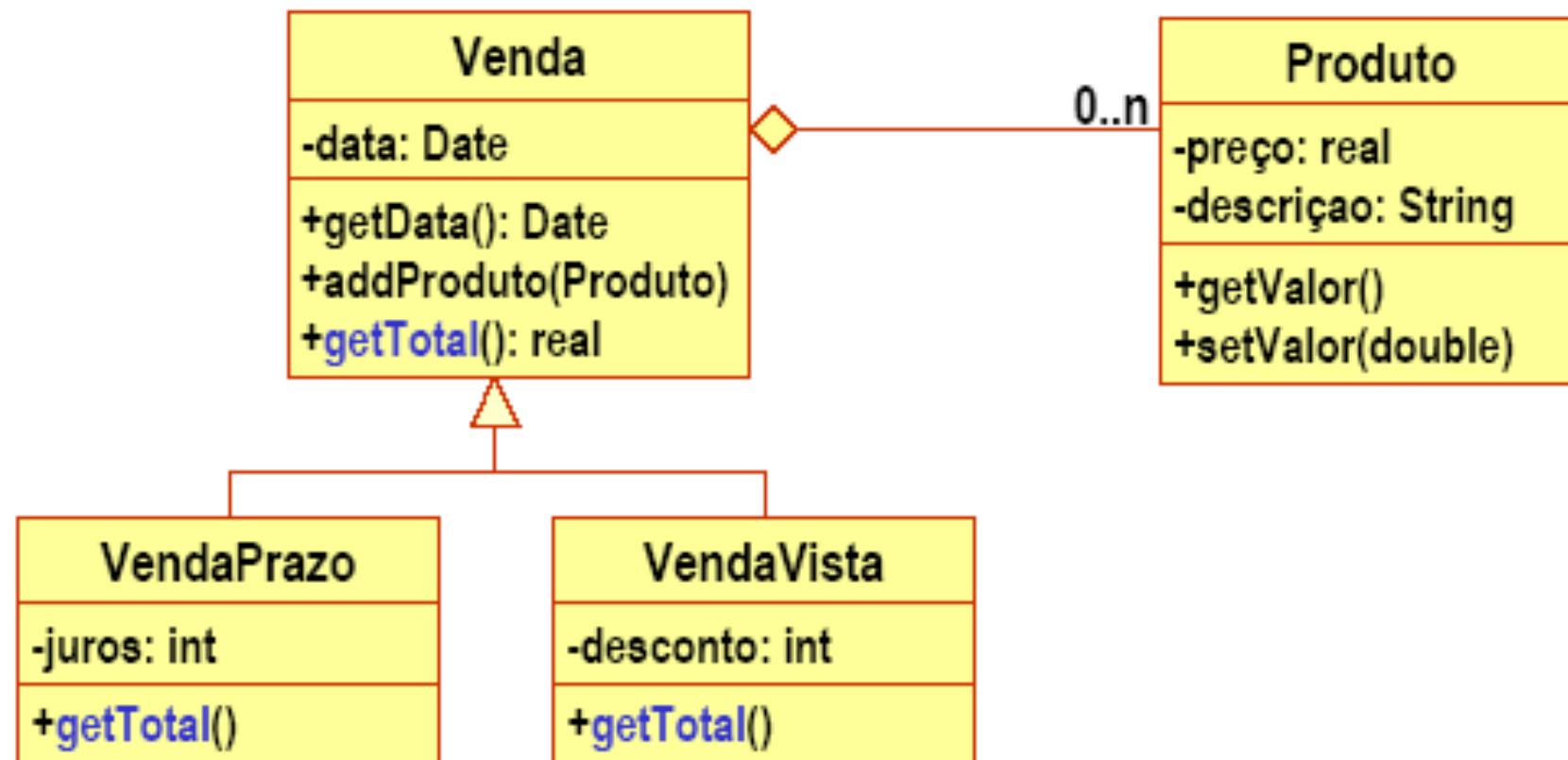
Herança (Revisão)

- Herdando o estado
 - Os objetos da subclasse herdam todas as propriedades declaradas em todas as suas superclasses
- Herdando o comportamento
 - Todos os métodos (que não sejam de classe) são herdados pela subclasse

Sobrescrita de Métodos

- Sobrescrita de métodos:
 - Se uma subclasse implementa um método que já existe na superclasse, dizemos que existe uma sobrescrita de métodos
 - A assinatura dos métodos deve ser a mesma
 - A busca pelo método inicia na subclasse, caso ele esteja marcado como `virtual`
 - O método sobrescrito pode chamar o método original indicando o nome da superclasse
- Cuidado para não confundir este conceito com o de sobrecarga

Sobrescrita de Métodos



Sobrescrita de Métodos

```
#ifndef PRODUTO_H
#define PRODUTO_H

// Classe para representar um produto
class Produto {
public:
    Produto(int codigo, int quantidade, double preco);
    double getTotal() const;

private:
    int codigo;
    int quantidade;
    double preco;
};

#endif
```

Sobrescrita de Métodos

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

Produto::Produto(int codigo, int quantidade, double preco) : codigo(codigo),
quantidade(quantidade), preco(preco) {}

double Produto::getTotal() const {
    return quantidade * preco;
}
```

Sobrescrita de Métodos

```
#ifndef VENDA_H
#define VENDA_H

#include <iostream>
#include <vector>
#include "Produto.h"
#include "Data.h"

class Venda {
public:
    Venda();
    // Método virtual puro para calcular o total
    virtual double getTotal() const;
    void adicionarProduto(const Produto& produto);
    virtual void imprime() const;
protected:
    double valorTotal;
    std::vector<Produto> produtos;
};

#endif
```

Sobrescrita de Métodos

```
#include "Venda.h"

Venda::Venda() {
    valorTotal=0.0;
}

double Venda::getTotal() const {
    return valorTotal;
}
// Método para adicionar um produto à venda
void Venda::adicionarProduto(const Produto& produto) {
    produtos.push_back(produto);
    valorTotal += produto.getTotal();
}

void Venda::imprime() const {
    std::cout << "Venda realizada no valor de: " << getTotal() << std::endl;
}
```

Sobrescrita de Métodos

```
#ifndef VENDA_VISTA_H
#define VENDA_VISTA_H

#include <iostream>
#include <vector>
#include "Venda.h"

class VendaVista: public Venda {
private:
    double desconto;
public:
    VendaVista(double);
    double getTotal() const override;
    void imprime() const override;
};

#endif
```

Sobrescrita de Métodos

```
#include "VendaVista.h"

VendaVista::VendaVista(double d) : Venda() {
    desconto=d;
}

double VendaVista::getTotal() const {
    return Venda::getTotal()-desconto;
}

void VendaVista::imprime() const {
    std::cout << "Venda a vista realizada no valor de: " << Venda::getTotal()
<< " com desconto de " << desconto << " totalizando " << getTotal()
<< std::endl;
}
```

Sobrescrita de Métodos

```
#ifndef VENDA_PRAZO_H
#define VENDA_PRAZO_H

#include <iostream>
#include <vector>
#include "Venda.h"

class VendaPrazo: public Venda {
private:
    double juros;
public:
    VendaPrazo(double);
    double getTotal() const override;
    void imprime() const override;
};

#endif
```

Sobrescrita de Métodos

```
#include "VendaPrazo.h"

VendaPrazo::VendaPrazo(double j): Venda(), juros(j) {}

double VendaPrazo::getTotal() const {
    return Venda::getTotal()*(1.0+(juros/100));
}

void VendaPrazo::imprime() const {
    std::cout << "Venda a prazo realizada no valor de: " << Venda::getTotal()
    << " com juros de " << juros << " totalizando " << getTotal() << std::endl;
}
```

Sobrescrita de Métodos

```
#ifndef CAIXA_H
#define CAIXA_H

#include <iostream>
#include <vector>
#include "Venda.h"

class Caixa {
private:
    std::vector<Venda*> vendas;
public:
    void adicionarVenda(Venda* venda);
    double calcularTotalVendas() const;
};

#endif
```

Sobrescrita de Métodos

```
#include "Caixa.h"

void Caixa::adicionarVenda(Venda* venda) {
    vendas.push_back(venda);
}

double Caixa::calcularTotalVendas() const {
    double total = 0.0;
    for (const Venda* venda : vendas) {
        total += venda->getTotal(); // Chama o getTotal da classe
correspondente (polimorfismo)
        venda->imprime();
    }
    return total;
}
```

Sobrescrita de Métodos

```
#include <iostream>
#include "Caixa.h"
#include "Produto.h"
#include "Venda.h"
#include "VendaVista.h"
#include "VendaPrazo.h"

int main()
{
    Venda *venda1,*venda2,*venda3;
    venda1 = new Venda();
    venda1->adicionarProduto(Produto(1215, 10, 9.50));

    venda2 = new VendaVista(15);
    venda2->adicionarProduto(Produto(1217, 1, 21.00));
    venda2->adicionarProduto(Produto(1218, 2, 24.00));

    venda3 = new VendaPrazo(10.0); // 10% de juros
    venda3->adicionarProduto(Produto(1223, 1, 22.00));
    venda3->adicionarProduto(Produto(1249, 3, 50.00));

    Caixa caixa;
    caixa.adicionarVenda(venda1);
    caixa.adicionarVenda(venda2);
    caixa.adicionarVenda(venda3);

    double totalVendas = caixa.calcularTotalVendas();

    std::cout << "Total de vendas no caixa: R$" << totalVendas << std::endl;
    return 0;
}
```

Venda realizada no valor de: 95
Venda a vista realizada no valor de: 69 com desconto de 15 totalizando 54
Venda a prazo realizada no valor de: 172 com juros de 10 totalizando 189.2
Total de vendas no caixa: R\$338.2

Sobrescrita de Métodos

```
double VendaPrazo::getTotal() const {  
    return Venda::getTotal() * (1.0 + (juros / 100));  
}
```

Método de VendaPrazo

Método de Venda

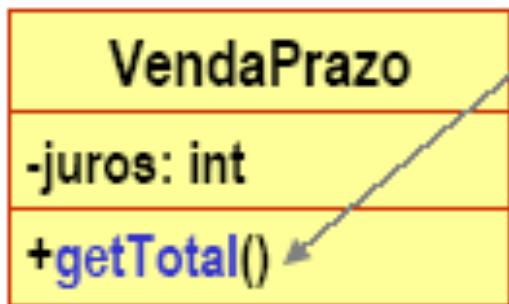
Sobrescrita de Métodos

- Uma mensagem só é ligada a um método em tempo de execução -> Ligação Dinâmica
 - Na ligação estática, a ligação entre a chamada de uma subrotina e sua implementação é estabelecida em tempo de compilação
- Com a herança, não se sabe exatamente o método que deve ser invocado até que se conheça o tipo do objeto (tipo dinâmico)
- Durante a compilação, só temos o tipo de referência (que é estático)

Sobrescrita de Métodos

```
double Caixa::calcularTotalVendas() const {
    double total = 0.0;
    for (const Venda* venda : vendas) {
        total += venda->getTotal();
        venda->imprime();
    }
    return total;
}
```

Qual getTotal() será chamado aqui?



Sobrescrita de Métodos

- A ligação entre a mensagem `venda->getTotal()` e o método `getTotal()` de uma das classes só é estabelecida em tempo de execução, quando o tipo do objeto em vendas for conhecido
 - A superclasse `Venda` é o tipo de referência de cada objeto do vetor
 - `VendaVista` e `VendaPrazo` são os tipos efetivos dos objetos instanciados
 - Exemplo:

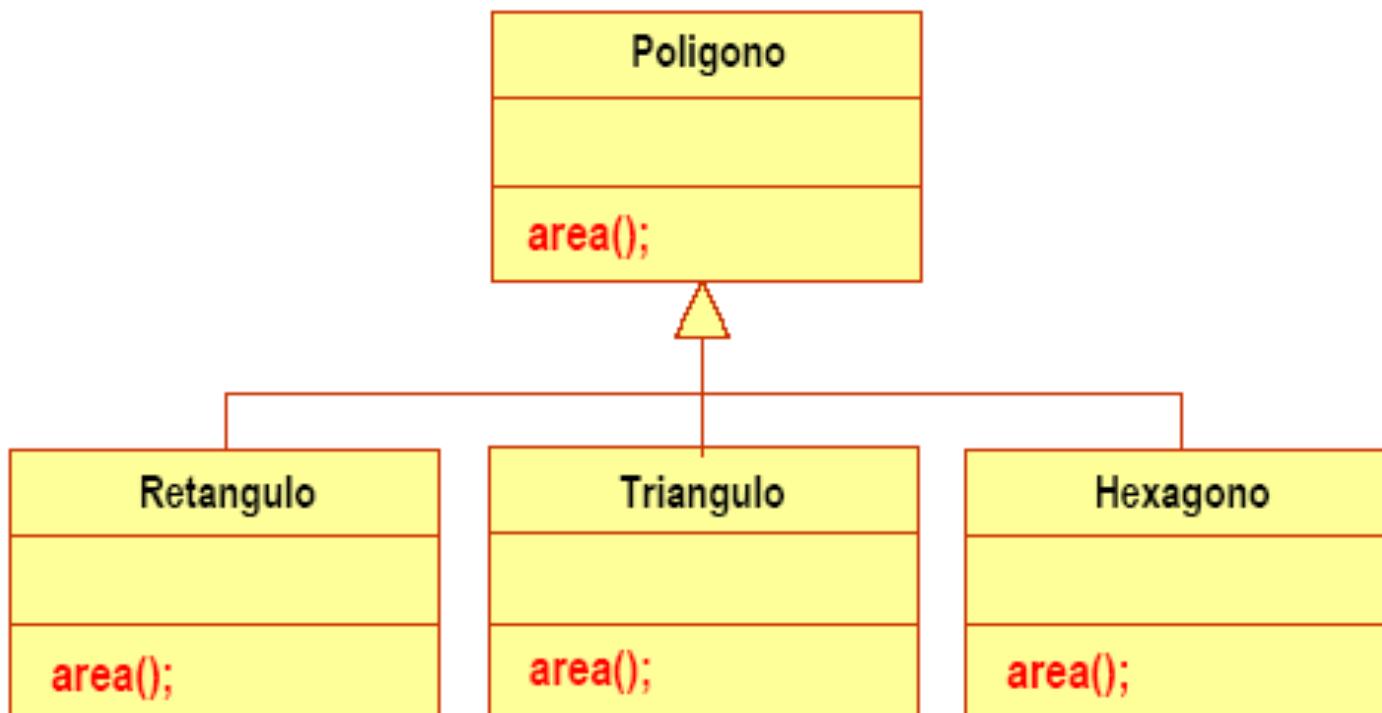
```
Venda *v = new VendaVista();
```

Polimorfismo

- Polimorfismo
 - Do Grego poly (muitas) + morpho (formas)
- Há dois aspectos importantes do polimorfismo
 - Métodos de mesmo nome são definidos em várias classes de uma hierarquia, podendo assumir diferentes implementações
 - Propriedade pela qual uma variável pode apontar para objetos de diferentes classes em momentos distintos

Polimorfismo

- Exemplo de polimorfismo nos métodos



Polimorfismo

- Exemplo de polimorfismo em variáveis
 - Polígono *p;
 - P = new Polígono();
 - ...
 - P = new Triângulo();
 - ...
 - P = new Retângulo();

Polimorfismo

```
#include <iostream>
class Poligono {
public:
    virtual void area() {
        std::cout << "Área do Polígono" << std::endl;
    }
};
class Triangulo : public Poligono {
public:
    virtual void area() override {
        std::cout << "Área do Triângulo" << std::endl;
    }
};
class Retangulo : public Poligono {
public:
    virtual void area() override {
        std::cout << "Área do Retângulo" << std::endl;
    }
};
int main() {
    Poligono* p;
    p = new Poligono();
    p->area(); // Chama o método da classe base
    p = new Triangulo();
    p->area(); // Chama o método da classe derivada Triangulo
    p = new Retangulo();
    p->area(); // Chama o método da classe derivada Retangulo
    return 0;
}
```

Uma vez que o **virtual** foi estabelecido na hierarquia, não é mais necessário repeti-lo

Área do Polígono
Área do Triângulo
Área do Retângulo

Sobrescrita sem virtual

```
#include <iostream>
class Poligono {
public:
    void area() {
        std::cout << "Área do Polígono" << std::endl;
    }
};
class Triangulo : public Poligono {
public:
    void area() override {
        std::cout << "Área do Triângulo" << std::endl;
    }
};
class Retangulo : public Poligono {
public:
    void area() override {
        std::cout << "Área do Retângulo" << std::endl;
    }
};
int main() {
    Poligono* p;
    p = new Poligono();
    p->area(); // Chama o método da classe base
    p = new Triangulo();
    p->area(); // Chama o método da classe base
    p = new Retangulo();
    p->area(); // Chama o método da classe base
    return 0;
}
```

Sem a palavra `virtual`, não a ligação dinâmica

Área do Polígono
Área do Polígono
Área do Polígono

Polimorfismo

- Legibilidade do Código
 - O mesmo nome para a mesma operação facilita a legibilidade do código
- Código de menor tamanho
 - Código mais claro, enxuto e elegante
 - Flexibilidade
 - Pode-se incluir novas classes sem alterar o código que irá manipulá-la
- **O polimorfismo é implementado com uso da ligação dinâmica**

Herança de operações e polimorfismo

- Uma subclasse herda todas as propriedades de sua superclasse que tenham visibilidade pública ou protegida.
- Entretanto, pode ser que o comportamento de alguma operação herdada seja diferente para a subclasse.
 - Nesse caso, a subclasse deve redefinir o comportamento da operação.
 - A assinatura da operação pode ser reutilizada.
 - A implementação da operação (**método**) é diferente.

Operações polimórficas

- Operações polimórficas são operações de mesma assinatura definidas em diversos níveis de uma hierarquia de generalização e que possuem comportamento diferente.
 - assinatura é repetida na(s) subclasse(s) para enfatizar a redefinição de implementação.
- Operações polimórficas implementam o princípio do polimorfismo no qual duas ou mais classes respondem a mesma mensagem de formas diferentes.
- Objetivo: garantir que as subclasses tenham uma interface em comum.

Operações abstratas e polimorfismo

- Em termos de operações, uma classe é abstrata quando ela possui pelo menos uma operação abstrata.
- Uma operação abstrata não possui implementação.
 - Uma classe pode possuir tanto operações abstratas quanto operações concretas.
 - Uma classe que possui pelo menos uma operação abstrata é abstrata.
- Uma subclasse que herda uma operação abstrata e não fornece uma implementação é ela própria abstrata.

Operações abstratas e polimorfismo

- **virtual:**

- `virtual` é usado para declarar uma função em uma classe base que pode ser substituída em classes derivadas.
- Deve ser usado na declaração da função na classe base, bem como em sua definição.
- Classes derivadas podem escolher se desejam ou não substituir a função `virtual`. Se não o fizerem, a implementação da classe base é usada.
- Uma função `virtual` permite que o mecanismo de ligação dinâmica (`dynamic binding`) seja ativado, permitindo que o método da classe derivada seja chamado em tempo de execução, com base no tipo real do objeto.

Operações abstratas e polimorfismo

- **override:**

- override é usado na declaração de uma função em uma classe derivada para indicar explicitamente que você pretende substituir uma função virtual da classe base.
- É uma boa prática incluir override ao definir funções em classes derivadas que devem substituir funções virtuais da classe base.
- Ajuda a evitar erros de digitação e outros erros comuns ao escrever código que se baseia em polimorfismo.

Classes e Métodos Abstratos

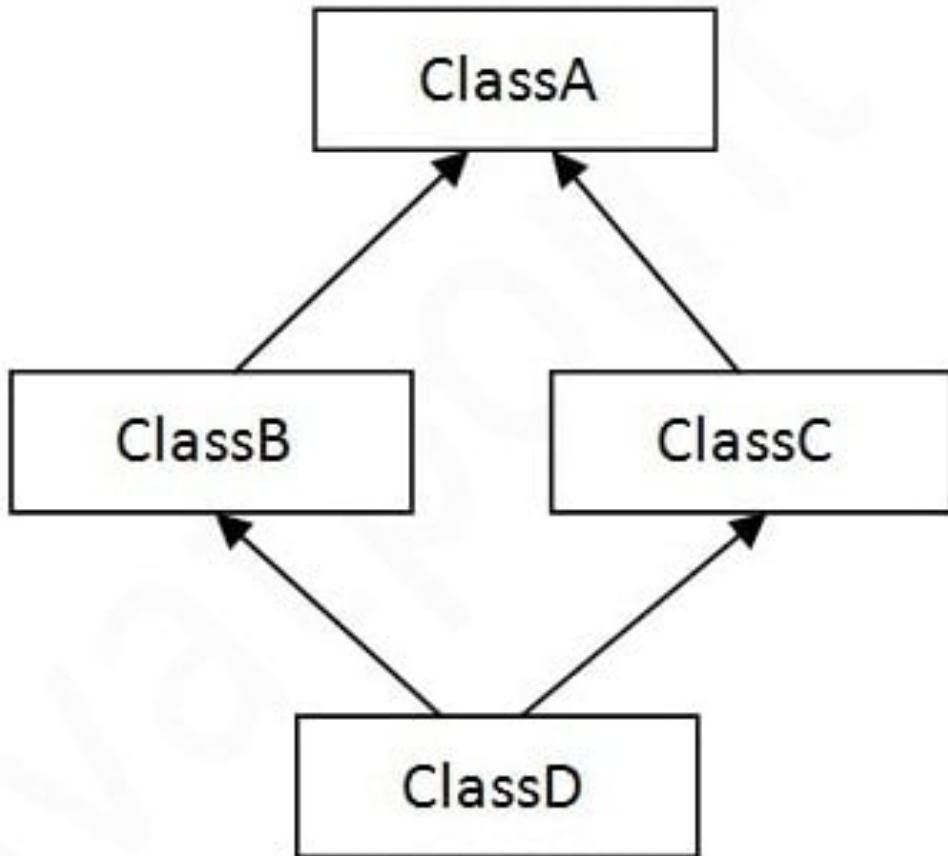
- Na declaração da classe usamos:

```
virtual void teste() =0;
```

- **Classes com métodos abstratos não podem ser instanciadas**
- Se as filhas da classe não implementam o método abstrato, elas também não podem ser instanciadas

Herança Múltipla

Herança Múltipla



Ambiguidades

- **Ambiguidade de nomenclatura:**

- Quando uma classe derivada herda dois membros com o mesmo nome de classes base diferentes, ocorre uma ambiguidade na chamada desses membros.
- Como resolver:
 - qualificação de escopo
 - sobrescrevendo o método

Exemplo

```
class A {  
public:  
    void mostrar() {  
        std::cout << "Classe A" << std::endl;  
    }  
};  
  
class B {  
public:  
    void mostrar() {  
        std::cout << "Classe B" << std::endl;  
    }  
};  
  
class C : public A, public B {  
};  
  
int main() {  
    C objetoC;  
    objetoC.mostrar(); // Ambiguidade: qual método mostrar() deve ser chamado?  
    return 0;  
}
```

Ambiguidades

- **Ambiguidade de conversão:**
 - Quando você herda de duas classes base que têm uma relação de herança comum, você pode enfrentar uma ambiguidade na conversão dessas classes base para a classe derivada

Exemplo

```
class A {  
public:  
    int valorA;  
};  
  
class B {  
public:  
    int valorB;  
};  
  
class C : public A, public B {  
public:  
    int valorC;  
};  
  
-----  
C objetoC;  
  
// Convertendo para a classe base A  
A* ponteiroA = (A*)&objetoC;  
ponteiroA->valorA = 10;  
  
// Convertendo para a classe base B  
B* ponteiroB = (B*)&objetoC;  
ponteiroB->valorB = 20;
```

Ambiguidades

- **Ambiguidade de função virtual:**

- Quando uma classe derivada herda duas funções virtuais com a mesma assinatura de classes base diferentes, pode ocorrer uma ambiguidade se a classe derivada não fornecer uma implementação própria para resolver a ambiguidade.
- Como resolver:
 - qualificação de escopo
 - sobrescrevendo o método

Exemplo

```
class A {  
public:  
    virtual void mostrar() {  
        std::cout << "Classe A" << std::endl;  
    }  
};  
  
class B {  
public:  
    virtual void mostrar() {  
        std::cout << "Classe B" << std::endl;  
    }  
};  
  
class C : public A, public B {  
};  
  
int main() {  
    C objetoC;  
    objetoC.mostrar(); // Ambiguidade: qual função mostrar() deve ser chamada?  
    return 0;  
}
```

Exemplo

```
class A {  
public:  
    virtual void mostrar() {  
        std::cout << "Classe A" << std::endl;  
    }  
};  
  
class B {  
public:  
    virtual void mostrar() {  
        std::cout << "Classe B" << std::endl;  
    }  
};  
  
class C : public A, public B {  
};  
  
int main() {  
    C objetoC;  
  
    objetoC.A::mostrar(); // Chama o método mostrar() da classe A  
    objetoC.B::mostrar(); // Chama o método mostrar() da classe B  
  
    return 0;  
}
```

Classe A
Classe B

Exemplo

```
#include <iostream>

// Primeira classe base
class A {
public:
    virtual void mostrar() {
        std::cout << "Classe A" << std::endl;
    }
};

// Segunda classe base
class B {
public:
    virtual void mostrar() {
        std::cout << "Classe B" << std::endl;
    }
};

// Classe derivada que herda de A e B
class C : public A, public B {
public:
    // Override da função mostrar para resolver ambiguidade
    void mostrar() override {
        std::cout << "Classe C" << std::endl;
    }
};
```

Exemplo

```
int main() {  
    C objetoC;  
  
    // Chama a função mostrar da classe C  
    objetoC.mostrar();  
  
    // Chama a função mostrar da classe A usando um ponteiro A*  
    A* ponteiroA = &objetoC;  
    ponteiroA->mostrar();  
  
    // Chama a função mostrar da classe B usando um ponteiro B*  
    B* ponteiroB = &objetoC;  
    ponteiroB->mostrar();  
  
    return 0;  
}
```

Classe C
Classe C
Classe C

Exemplo

```
#include <iostream>

// Primeira classe base
class A {
public:
    void mostrar() {
        std::cout << "Classe A" << std::endl;
    }
};

// Segunda classe base
class B {
public:
    void mostrar() {
        std::cout << "Classe B" << std::endl;
    }
};

// Classe derivada que herda de A e B
class C : public A, public B {
public:
    void mostrar() {
        std::cout << "Classe C" << std::endl;
    }
};
```

E se removermos o **virtual**?

Exemplo

```
int main() {  
    C objetoC;  
  
    // Chama a função mostrar da classe C  
    objetoC.mostrar();  
  
    // Chama a função mostrar da classe A usando um ponteiro A*  
    A* ponteiroA = &objetoC;  
    ponteiroA->mostrar();  
  
    // Chama a função mostrar da classe B usando um ponteiro B*  
    B* ponteiroB = &objetoC;  
    ponteiroB->mostrar();  
  
    return 0;  
}
```

Classe C
Classe A
Classe B

Herança Virtual

- Permite evitar a criação de múltiplas cópias de classes base compartilhadas por classes derivadas, resolvendo assim os problemas de ambiguidade

```
#include <iostream>

class Animal {
public:
    void comer() {
        std::cout << "O animal está comendo." << std::endl;
    }
};

class Peixe : public Animal {
public:
    void nadar() {
        std::cout << "O peixe está nadando." << std::endl;
    }
};
```

Herança Virtual

- Permite evitar a criação de múltiplas cópias de classes base compartilhadas por classes derivadas, resolvendo assim os problemas de ambiguidade

```
class Ave : public Animal {  
public:  
    void voar() {  
        std::cout << "A ave está voando." << std::endl;  
    }  
};  
  
class Pinguim : public Peixe, public Ave {  
};  
  
int main() {  
    Pinguim pinguim;  
    pinguim.comer(); // Ambiguidade: qual comer() deve ser chamado?  
    return 0;  
}
```

Herança Virtual

```
#include <iostream>
class Animal {
public:
    void comer() {
        std::cout << "O animal está comendo." << std::endl;
    }
};

class Peixe : virtual public Animal {
public:
    void nadar() {
        std::cout << "O peixe está nadando." << std::endl;
    }
};

class Ave : virtual public Animal {
public:
    void voar() {
        std::cout << "A ave está voando." << std::endl;
    }
};

class Pinguim : public Peixe, public Ave {
};

int main() {
    Pinguim pinguim;
    pinguim.comer(); // Chama a versão de comer() da classe Animal
    return 0;
}
```

Ponteiro de Métodos

Ponteiros de Métodos

- Ponteiros de métodos são utilizados em linguagens OO para permitir a chamada dinâmica de métodos de objetos em tempo de execução
- Eles são uma parte fundamental do polimorfismo em C++ e em outras linguagens que suportam essa característica.
- **Sintaxe:**

```
tipo_de_retorno (Classe::*nome_do_ponteiro_de_metodo) (lista_de_argumentos)
    • tipo_de_retorno: O tipo de retorno do método que o ponteiro de método aponta.
    • Classe: O nome da classe que contém o método.
    • nome_do_ponteiro_de_metodo: O nome que você escolhe para o ponteiro de método.
    • lista_de_argumentos: A lista de argumentos do método que o ponteiro de método aponta.
```

- **Exemplo de declaração:**

```
int (MinhaClasse::*ponteiroMetodo) (int, double);
```

- **Exemplo de uso:**

```
MinhaClasse objeto;
*ponteiroMetodo = &objeto.metodoEscolhido;
int resultado = (objeto.*ponteiroMetodo) (5, 3.14);
```

- Se o método for estático, o ponteiro é declarado como um ponteiro de função, sem referência a classe

Ponteiros de Métodos

- **Alguns exemplos de uso de ponteiros de método:**

- Polimorfismo Os ponteiros de método são frequentemente usados para implementar o polimorfismo. Você pode criar um ponteiro de método que aponte para um método de uma classe base e, em seguida, usá-lo para chamar dinamicamente o método de uma classe derivada. Isso permite que diferentes objetos usem métodos diferentes com a mesma interface.
- Máquinas de estado: Em máquinas de estado finito ou máquinas de eventos, você pode usar tabelas ou arrays de ponteiros de método para representar as transições de estado ou eventos. Isso torna a implementação de máquinas de estado mais flexível e fácil de manter.
- Despacho de eventos: Em sistemas que lidam com eventos, como interfaces gráficas de usuário, você pode usar ponteiros de método para associar eventos a manipuladores de eventos específicos. Isso permite que objetos respondam a eventos de forma dinâmica.
- Implementação de padrões de projeto: Ponteiros de método são frequentemente usados na implementação de padrões de projeto como o Strategy, onde diferentes algoritmos podem ser encapsulados em classes separadas e selecionados em tempo de execução com base em um ponteiro de método.
- Interfaces abstratas: Em algumas implementações de interfaces abstratas, os ponteiros de método são usados para representar os métodos que as classes derivadas devem implementar. Isso permite que você defina contratos de interface flexíveis.
- Testes e Mocking: Ponteiros de método são úteis para criar mocks e simulações em testes de unidade, permitindo que você substitua temporariamente a implementação de métodos para fins de teste.

Ponteiro de método

```
#include <iostream>
class Animal {
public:
    virtual void emitirSom() {
        std::cout << "Animal emite um som genérico" << std::endl;
    }
};

class Cachorro : public Animal {
public:
    void emitirSom() override {
        std::cout << "Cachorro late" << std::endl;
    }
};

class Gato : public Animal {
public:
    void emitirSom() override {
        std::cout << "Gato mia" << std::endl;
    }
};
```

Ponteiro de método

```
int main() {
    Animal* animalPtr = nullptr; // Ponteiro para a classe base

    Animal animal;
    Cachorro cachorro;
    Gato gato;

    // Ponteiro para o método emitirSom da classe base
    void (Animal::*ptrEmitirSom) () = &Animal::emitirSom;

    // Apontar para o método emitirSom de Cachorro
    animalPtr = &cachorro;
    (animalPtr->*ptrEmitirSom) (); // Chama o método Cachorro::emitirSom()

    // Apontar para o método emitirSom de Gato
    animalPtr = &gato;
    (animalPtr->*ptrEmitirSom) (); // Chama o método Gato::emitirSom()

    // Apontar para o método emitirSom da classe base
    animalPtr = &animal;
    (animalPtr->*ptrEmitirSom) (); // Chama o método Animal::emitirSom()

    return 0;
}
```

Cachorro late
Gato mia
Animal emite um som genérico

Herança x Associação

Herança x Associação

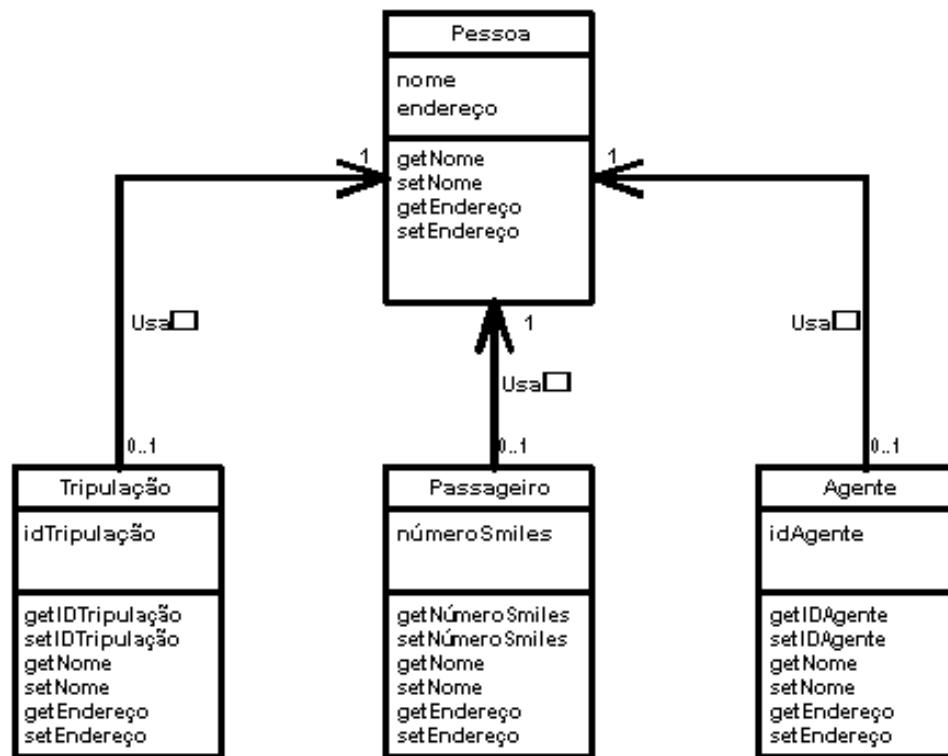
- Herança (Vantagens)
 - Reutilização de código
 - Polimorfismo
 - Estrutura Hierárquica
 - Substituição de método

Herança x Associação

- Associação (Vantagens)
 - Flexibilidade
 - mecanismo de acoplamento mais flexível do que a herança. Permite que objetos de diferentes classes sejam relacionados sem impor uma hierarquia estrita
 - Separação de preocupações: a associação permite separar diferentes preocupações ou responsabilidades em classes distintas.
 - Relacionamentos Complexos
 - Minimizando Dependências
 - Criação de objetos em tempo de execução

Herança x Associação

- Considere aplicar padrões de projeto como o princípio Composição sobre Herança (Col) ao lidar com relacionamentos complexos.
- Col promove o uso de composição (associação) para construir objetos com comportamentos desejados, em vez de depender de herança.



Herança x Associação

- Estamos estendendo a funcionalidade de Pessoa de várias formas, mas sem usar herança
- Observe que também podemos inverter a composição (uma pessoa tem um ou mais papéis)
 - Pense na implicação para a interface de "pessoa"
- Aqui, estamos usando delegação: dois objetos estão envolvidos em atender um pedido (digamos setNome)
 - O objeto tripulação (digamos) delega setNome para o objeto pessoa que ele tem por composição
 - Técnica também chamada de forwarding
 - É semelhante a uma subclasse delegar uma operação para a superclasse (herdando a operação)
 - Delegação sempre pode ser usada para substituir a herança
 - Se usássemos herança, o objeto tripulação poderia referenciar a pessoa com this
 - Com o uso de delegação, tripulação pode passar this para Pessoa e o objeto Pessoa pode referenciar o objeto original se quiser
 - Em vez de tripulação ser uma pessoa, ele tem uma pessoa
 - A grande vantagem da delegação é que o comportamento pode ser escolhido em tempo de execução e vez de estar amarrado em tempo de compilação
 - A grande desvantagem é que um software muito dinâmico e parametrizado é mais difícil de entender do que software mais estático

- Próxima aula
 - Módulo 3 – recursos avançados de C++