

# IMD0030

# LINGUAGEM DE PROGRAMAÇÃO I

Aula 16 – Herança

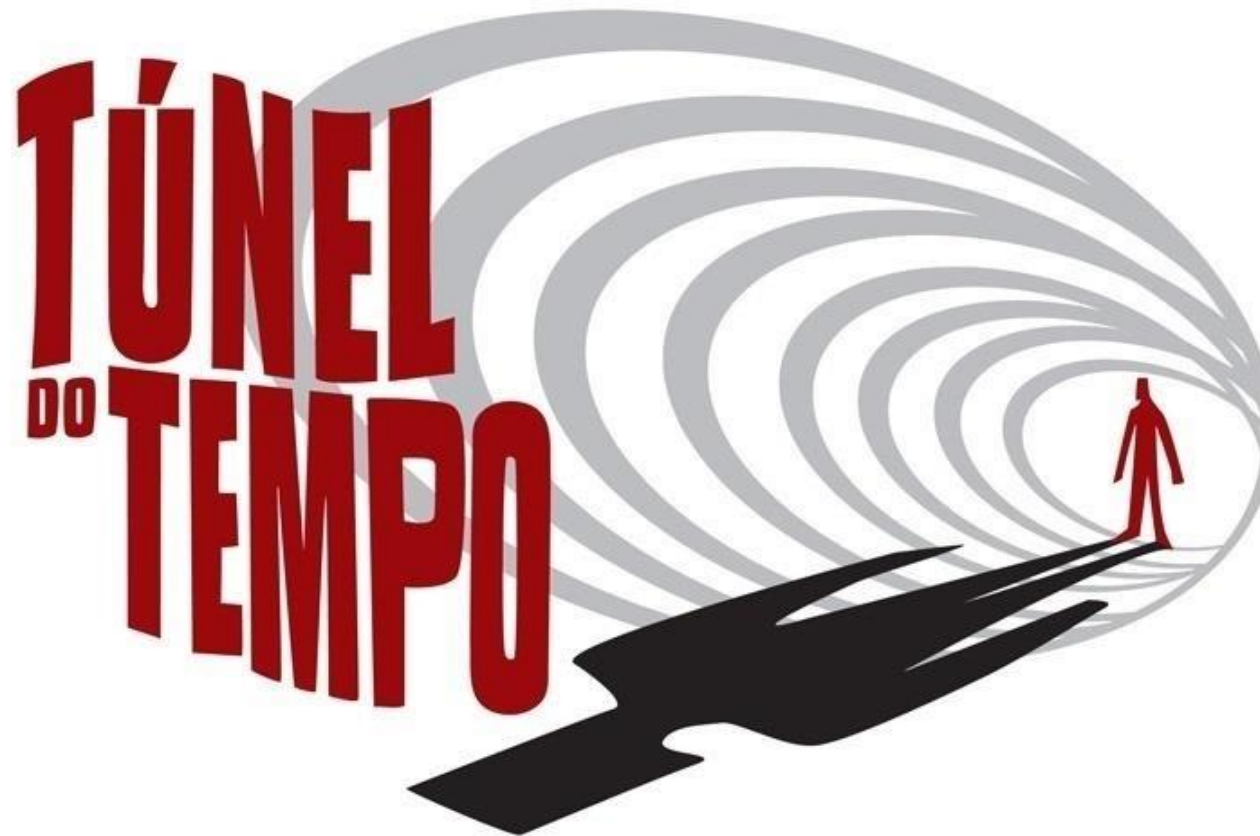
---

# Objetivos desta aula

- Introduzir o conceito de herança na linguagem de programação C++
- Para isso, estudaremos:
  - O que é herança e qual o seu papel no paradigma de Programação Orientada a Objetos (POO)
  - Como implementar classes que herdem membros (atributos e/ou métodos) de outras classes
- Ao final da aula, espera-se que o aluno seja capaz de:
  - Compreender o conceito de herança
  - Implementar classes na linguagem de programação C++ que façam uso do mecanismo de herança

---

Nas cenas dos capítulos anteriores...



---

# Nas cenas dos capítulos anteriores...

- Vimos que o **paradigma de Programação Orientada a Objetos (POO)** surgiu com o objetivo principal de facilitar o desenvolvimento de programas
  - agregando novos conceitos para a representação de elementos do mundo real de forma mais intuitiva
  - procurando melhorar produtividade e qualidade no desenvolvimento de *software*
- Vimos que a solução de problemas utilizando POO é essencialmente baseada na **abstração**
  - das **entidades** do mundo real a serem representadas no programa
  - dos **dados e características** associados a tais entidades
  - das **ações** que podem ser realizadas por tais entidades

# Nas cenas dos capítulos anteriores...

Vimos que, em POO,

- **classes** representam entidades do mundo real
- **objetos** são instâncias de classes e representam indivíduos de uma entidade
- dados e características associados a um objeto são representados como **atributos** de classes
- as ações que podem ser realizadas por um objeto são implementadas como **métodos** de classes

Conceito	Elemento do mundo real
Entidade	Carro
Indivíduos	Carro X, carro Y, Carro Z, etc.
Dados e características	Cor, modelo, ano, placa, proprietário
Ações	Ligar, andar, parar

```
class Carro {
    string cor;
    string modelo;
    string ano;
    string placa;
    Pessoa proprietario;
    void ligar();
    void andar();
    void parar();
};

int main {
    Carro x;
    Carro y;
    Carro z;

    x.ligar();
    x.andar();
    x.parar();
};
```

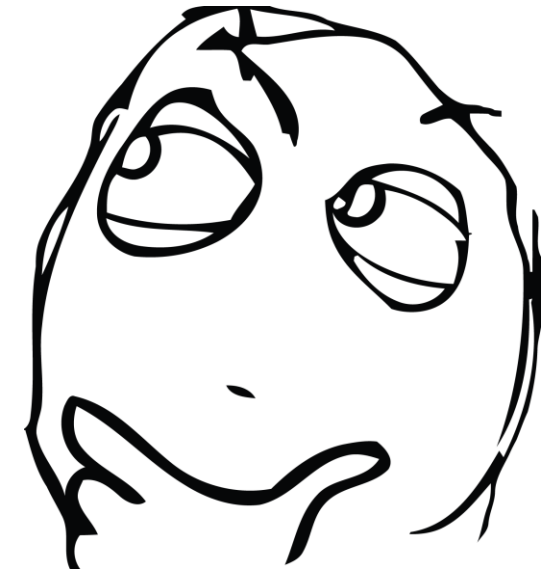
---

# Mas e se...

...quiséssemos criar classes para representar um caminhão e uma moto, de forma similar a um carro?

```
class Caminhao {  
    string cor;  
    string modelo;  
    string ano;  
    string placa;  
    Pessoa proprietario;  
    double capacidadeKg;  
    int qtdeEixos;  
    void ligar();  
    void andar();  
    void parar();  
};
```

```
class Moto {  
    string cor;  
    string modelo;  
    string ano;  
    string placa;  
    Pessoa proprietario;  
    double cilindradas;  
    void ligar();  
    void andar();  
    void parar();  
};
```



Qual o problema dessa abordagem?

As classes Carro, Caminhao e Moto possuem **membros idênticos**, havendo **repetição de código**

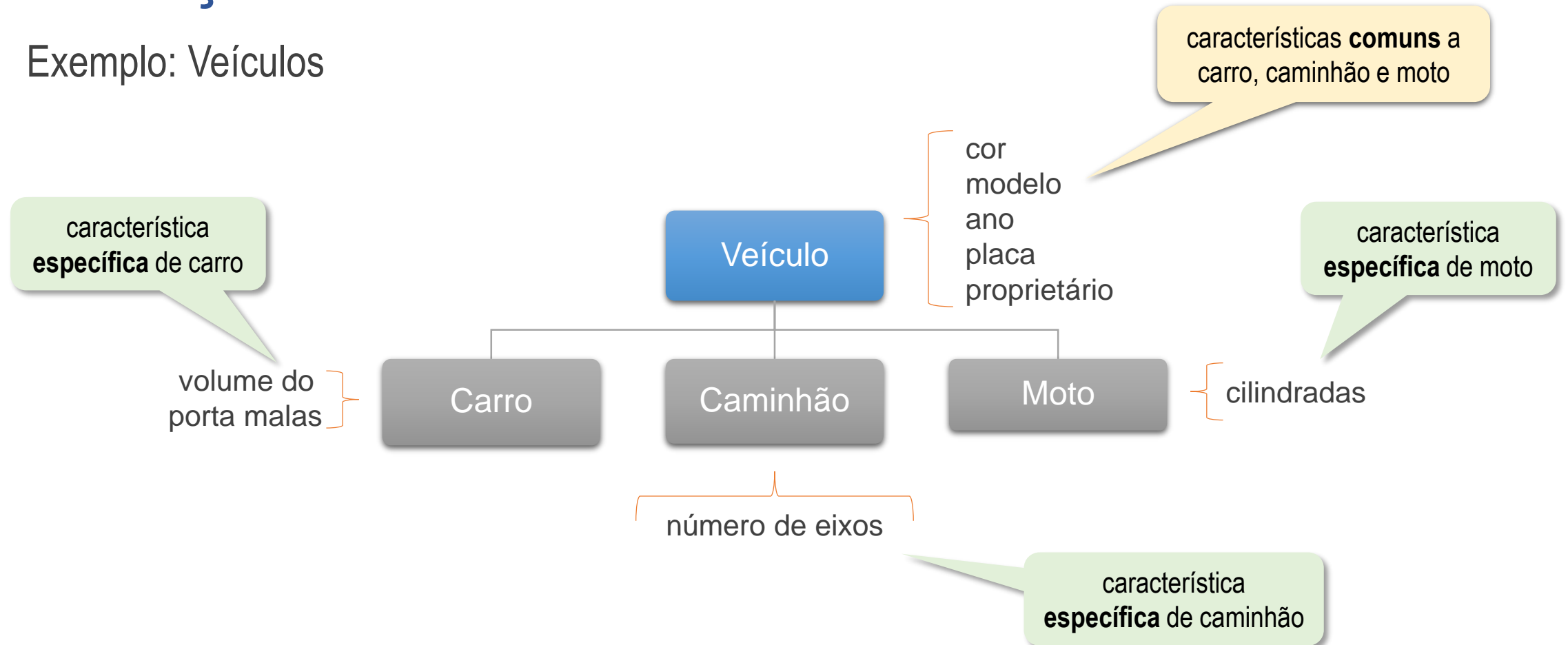
---

# Herança

- Mecanismo existente em POO que permite que uma classe **herde** membros (atributos e/ou métodos) de outra classe
  - Torna o conceito de classe mais poderoso
- Objetivos: aumentar **reuso**, produtividade e simplicidade na programação
- Na herança
  - membros comuns a diferentes classes são reunidos em uma única classe, conhecida como **classe base** ou **superclasse**
  - a partir da classe base, outras classes (chamadas **classes derivadas** ou **subclasses**) podem ser definidas possuindo os mesmos membros especificados na classe base
  - as classes derivadas podem conter membros que sejam **particulares** a elas, ou seja, não são compartilhados com as outras classes derivadas

# Herança

Exemplo: Veículos

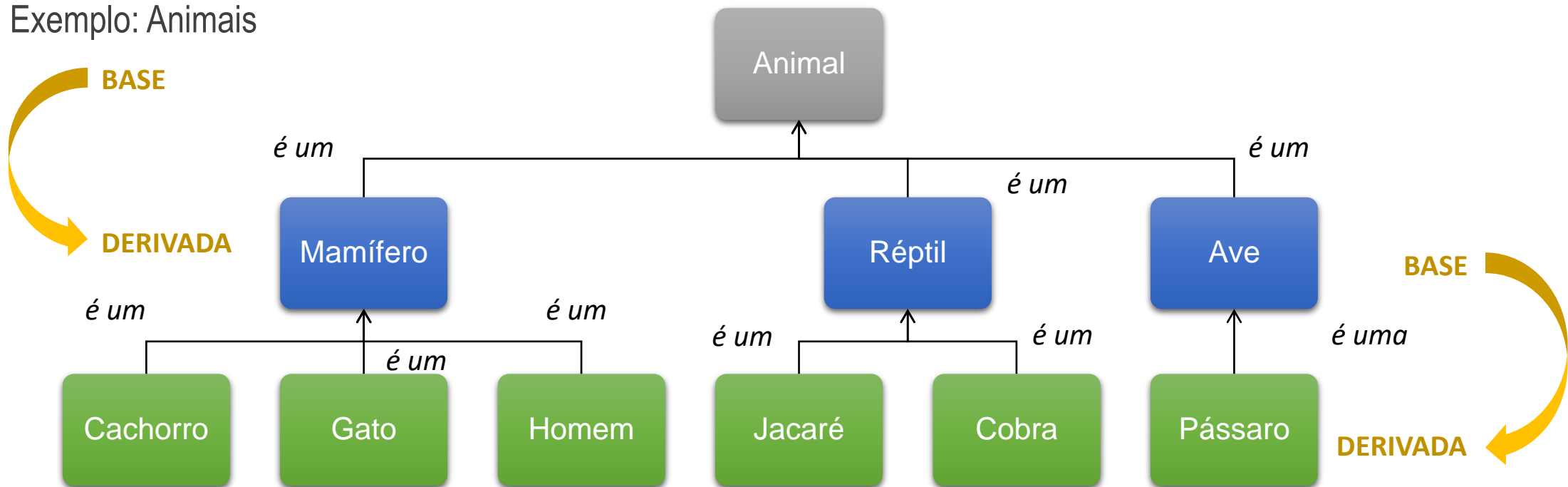




# Herança

O relacionamento entre objetos de classes base e de classes derivadas é tipicamente chamado de **é-um**

Exemplo: Animais



---

# Como implementar herança entre classes em C++

- A classe base é **implementada como uma classe comum**
  - Uma classe derivada pode também ser uma classe base
- Além dos modificadores de acesso já apresentados (*public* e *private*), existe um outro especificador relacionado estritamente ao conceito de herança: ***protected***
  - Membros ***protected*** são visíveis às classes derivadas, enquanto que membros privados (*private*) não
  - Se os membros forem definidos como privados (*private*), **apenas os métodos da classe base** terão acesso a eles

```
#include <string>
using std::string;

#include "pessoa.h"

class Veiculo {
    protected:
        string cor;
        string modelo;
        string ano;
        string placa;
        Pessoa proprietario;
    public:
        void ligar();
        void andar();
        void parar();
};
```

---

# Como implementar herança entre classes em C++

- A criação de classes derivadas que herdam de uma classe base é feita acrescentando-se
  - um modificador de acesso (tipicamente *public*)
  - o operador : (dois-pontos)
- As classes derivadas podem ter **novos atributos e métodos** além dos já existentes na classe base e que foram herdados
  - Os membros públicos e protegidos definidos na classe base são herdados pela classe derivada

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

using namespace std;

class Carro : public Veiculo {
    private:
        double volumePortaMalas;
    public:
        string getVolumePortaMalas();
        void setVolumePortaMalas(double v);
        void ligar();
        void andar();
        void parar();
};

void Carro::ligar() {
    cout << "Carro foi ligado" << endl;
}
```

---

# Como implementar herança entre classes em C++

- As classes derivadas podem **sobrescrever** métodos definidos na classe base
  - A classe derivada passa **redefine a implementação** do método
- O método tem de ter **exatamente a mesma assinatura**
  - Se um objeto da classe base invoca o método, é executada a versão da classe base
  - Se um objeto da classe derivada invoca o método, é executada a versão da classe derivada

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

using namespace std;

class Carro : public Veiculo {
    private:
        double volumePortaMalas;
    public:
        string getVolumePortaMalas();
        void setVolumePortaMalas(double v);
        void ligar();
        void andar();
        void parar();
};

void Carro::ligar() {
    cout << "Carro foi ligado" << endl;
}
```

---

# Como implementar herança entre classes em C++

- Resumo dos modificadores de acesso aplicadas aos membros de uma classe

Tipo de acesso	<i>public</i>	<i>private</i>	<i>protected</i>
Membros da mesma classe	SIM	SIM	SIM
Membros de classes derivadas	SIM	NÃO	SIM
Não membros	SIM	NÃO	NÃO

- Modificadores de acesso também são aplicados à classe base

```
class classeDerivada : <tipo_acesso> classeBase { ... };
```

- ***public*** : não altera a visibilidade dos membros da classe
- ***private***: herda os membros públicos (***public***) e protegidos (***protected***) como privados
- ***protected***: herda os membros públicos (***public***) e protegidos (***protected***) como protegidos

# Como implementar herança entre classes em C++

- A instanciação de objetos de classes derivadas é feita normalmente
  - Acesso a atributos e métodos da própria classe
  - Acesso a atributos e métodos da classe base com visibilidade protegida

```
int main() {  
    Carro c;  
    c.setModelo("Toyota Corolla");  
    c.setPlaca("ABC-1234");  
    c.setVolumePortaMalas(22.50);  
    c.ligar();  
  
    return 0;  
}
```

membros definidos na  
classe base Veiculo

membro específico da  
classe derivada Carro

método da classe base Veiculo  
**sobrescrito** pela classe derivada Carro

---

# Upcasting e downcasting

- Upcasting e downcasting são conceitos importantes do C++ e possibilitam a criação de programas complexos, mas mantendo uma sintaxe simples
  - Isto é conseguido através do Polimorfismo
- C++ permite que um ponteiro (ou referencia) para uma classe derivada seja tratado como um ponteiro para a classe base
  - Isso é upcasting
- Downcasting é o processo oposto, no qual um ponteiro (ou referência) para a classe base é convertido para um ponteiro para a classe derivada

---

# Upcasting

- Quando é feito um upcasting, o objeto não é modificado
  - Assim, quando um upcast é realizado, **será somente possível acessar membros que estão definidos na classe base**

```
int main() {  
    Veiculo* v = new Carro;  
    v->setModelo("Toyota Corolla");  
    v->setPlaca("ABC-1234");  
    v->setVolumePortaMalas (22.50); // ERRO: Pois upcasting foi utilizado  
    v->ligar();  
  
    return 0;  
}
```



---

# Downcasting

- Downcasting não é seguro como um upcasting
  - No upcasting há a garantia de que um objeto da classe derivada pode sempre ser tratada como um objeto da classe base, uma vez que todas as classes derivadas herdam os mesmos membros da classe base
  - Entretanto, no caso oposto (downcasting) não se pode dizer o mesmo
    - No exemplo aqui usado: Todo Carro é um Veículo, mas nem todo Veículo é um Carro

```
Veiculo* v = new Veiculo;
```

```
Carro* c1 = (Carro*) (v);  
c1->setModelo("Toyota Corolla");  
c1->setPlaca("ABC-1234");  
c1->setVolumePortaMalas(22.50);  
c1->ligar();
```

---

# Typecasting no C++

- Em C, quando se pretende converter uma expressão e num tipo T, utiliza-se a notação **(T)e** ou **T(e)**
  - Exemplo: (int) x OU int(x)
- O C++ introduz novos operadores para a conversão de tipos (cast):
  - **static\_cast**
    - Verifica a validade da conversão em tempo de compilação
  - **dynamic\_cast**
    - Verifica a validade da conversão em tempo de execução
  - **const\_cast**
    - Sobrecarrega a definição *const* durante a conversão
  - **reinterpret\_cast**
    - Conversão entre tipos não relacionados

---

# static\_cast

- O operador **static\_cast** realiza conversões entre tipos relacionados tais como:
  - conversão entre ponteiros dos tipos pertencentes a mesma hierarquia de classes (ou de um ponteiro do tipo *void* para qualquer outro ponteiro);
  - conversão de um tipo enumerado num tipo inteiro;
  - conversão do tipo *double* num tipo inteiro;
  - conversões que podem ser feitas implicitamente (conversão automática entre tipos).

- Exemplo:

```
class CData
{   public:
        static enum DiaSemana { segunda = 0, terca, quarta, quinta,
                                sexta, sabado, domingo };
};
```

```
int day = 3;
CData::DiaSemana ds = day; //erro
CData::DiaSemana ds = static_cast<CData::DiaSemana>(day);
```

---

# dynamic\_cast

- Operador que permite conversão do tipo base para o tipo derivado (downcasting)
  - Para efetuar um *cast* seguro usa-se o operador **dynamic\_cast**
  - Este operador recebe uma referência ou um ponteiro para um tipo polimórfico (**a classe deve incluir funções virtuais**)
  - Deve ser usado quando a correção da conversão não pode ser determinada pelo compilador
  - O operador **dynamic\_cast** pode converter uma classe base virtual polimórfica numa classe derivada (downcasting).

```
Veiculo* v = new Veiculo;  
  
Carro* c1 = dynamic_cast<Carro*>(v); // Funciona apenas se Veiculo possuir  
                                     // algum método virtual  
c1->setModelo("Toyota Corolla");  
c1->setPlaca("ABC-1234");  
c1->setVolumePortaMalas(22.50);  
c1->ligar();
```

---

# reinterpret\_cast

- O operador **reinterpret\_cast** realiza conversões entre tipos não relacionados
  - Normalmente, o operador produz valor de um tipo novo composto por mesmos bits que o argumento
  - O resultado só é utilizável se o tipo resultante coincide exatamente com o tipo do argumento
  - O **reinterpret\_cast** é recomendado apenas para operações onde se deseja converter um tipo básico para ponteiro e vice-versa

```
void imprimeCPF (Objeto *obj)
{
    Pessoa *p = reinterpret_cast<Pessoa *>(obj);

    std::cout << p->getCPF() << std::endl;
}
```

---

# const\_cast

- O operador **const\_cast** serve para remover os qualificadores *const* e *volatile* quando necessário

```
const int i = 0;

int* j = &i; // erro

int* j = (int*)&i; // sintaxe do C; não aconselhada

j = const_cast<int*>(&i); // esta sintaxe é melhor
```

---

# Cast com uso de ponteiros inteligentes

- Como os ponteiros inteligentes não são simples ponteiros, mas classes que gerenciam ponteiros, o simples *cast* não pode ser utilizado
- Para permitir o *cast* entre ponteiros inteligente o C++ fornece os operadores:
  - `std::static_pointer_cast`
  - `std::dynamic_pointer_cast`
  - `std::const_pointer_cast`

---

# Como implementar herança entre classes em C++

**Nem tudo é herdado** quando se declara uma classe derivada:

- Construtores
- Destrutores
- Relacionamentos *friend*
- Atributos com visibilidade privada (*private*)



---


# Métodos construtores e destrutores em herança

- Se a classe for derivada de alguma outra, o método construtor da classe base é invocado **antes** do método construtor da classe derivada
- Se a classe base também é derivada de outra, o processo é repetido recursivamente até que uma classe base não derivada seja alcançada
- Se uma classe base não possui um método construtor padrão, a classe derivada deve **obrigatoriamente** definir um método construtor padrão, ainda que vazio
- No caso dos métodos destrutores, a ordem de chamada é invertida: invoca-se primeiro o método destrutor da classe derivada e depois o método destrutor da classe base

# Herança múltipla

- Os tipos de herança vistos até então são chamados de **herança simples**: a classe derivada herda de apenas uma classe base
- A linguagem C++ permite realizar **herança múltipla**: uma mesma classe derivada pode herdar de mais de uma classe ao mesmo tempo
  - Após o operador : (dois-pontos), segue lista com nomes das classes base das quais a classe derivada irá herdar todos os atributos e métodos públicos ou protegidos

```
class BemMove1 {  
    protected:  
        float preco;  
        string codReceita;  
};  
  
class Veiculo {  
    protected:  
        string cor;  
        string modelo;  
        string ano;  
        string placa;  
        Pessoa proprietario;  
};  
  
class Carro : public Veiculo, BemMove1 {  
    protected:  
        string classificacao;  
    public:  
        // metodos da classe  
};
```

Two orange arrows point from the 'Veiculo' and 'BemMove1' class definitions to the 'Carro' class definition, indicating that 'Carro' inherits from both.

# Alguma Questão?

