>>> Programação Orientada a Objetos (POO)

... Herança

Prof: André de Freitas Smaira

[~]$ _

>>> Mais de construtores...

[~]$ _

>>> Construtor de movimento

* Quando passamos um **parâmetro por valor => construtor de cópia** (cópia completa dos dados)
* **Custoso** quando se copia os valores
* **Inseguro** quando se copia ponteiro
* Mas e se o objeto original não for mais usar aquele dado?
* => **Construtor de Movimento**
* **Move** os recursos de um objeto para outro, **sem cópias**
* Usado para **objetos temporários**

>>> Exemplo

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

class MeuObjeto {
public:
    std::string* data;  // Ponteiro para simular recurso dinâmico

    MeuObjeto(const std::string& valor) : data(new std::string(valor)) {
        std::cout << "Construtor padrão: " << *data << "\n";
    }

    MeuObjeto(MeuObjeto&& outro) noexcept : data(outro.data) {
        outro.data = nullptr;  // Esvazia o objeto original
        std::cout << "(Construtor de movimento)\n";
    }

    ~MeuObjeto() {
        if (data) {
            std::cout << "Destrutor: liberando " << *data << "\n";
            delete data;
        } else {
            std::cout << "Destrutor: nada para liberar\n";
        }
    }

    void verificaEstado() const {
        if (data) {
            std::cout << "Objeto contém: " << *data << "\n";
        } else {
            std::cout << "Objeto está vazio (nullptr)\n";
        }
    }
};
```

[~]$ _

>>> Exemplo

```cpp
int main() {
    // Criando o primeiro objeto
    MeuObjeto obj1("Teste de movimento");

    // Verificando o estado do primeiro objeto
    std::cout << "\nEstado de obj1 antes do movimento:\n";
    obj1.verificaEstado();

    // Movendo o conteúdo de obj1 para obj2
    MeuObjeto obj2 = std::move(obj1);  // Construtor de movimento é

    // Verificando o estado de obj1 após a movimentação
    std::cout << "\nEstado de obj1 após o movimento:\n";
    obj1.verificaEstado();

    // Verificando o estado de obj2 após a movimentação
    std::cout << "\nEstado de obj2 após o movimento:\n";
    obj2.verificaEstado();

    return 0;
}
```

]$ _

>>> Herança

[~]$ _

>>> Relações

* Classes são muitas vezes relacionadas
* Composição (tem-um) ✓
* Especialização (é-um): Herança

[~]$ _

>>> Herança

* Classe A pode herdar da classe B se **A é um B**. Exemplos:
* **Funcionário** é uma **Pessoa**
* **Quadrado** é-um **retângulo**
* **Pick-up** é-um **Automóvel**
* **Pisca-pisca** é-uma **lâmpada**

[~]$ _

>>> Herança

* **Classe base** (ou **superclasse**): mais geral
* **Classe derivada** (ou **subclasse**): mais específica
* **Membros da classe base** existem na derivada
* **Classe derivada** pode acrescentar **novos membros**
* Classe derivada **pode alterar métodos** herdados

>>> **Exemplo**

```
class Veiculo
{
    int _numero_eixos;
    double _potencia;
public:
    Veiculo() : _numero_eixos(2), _potencia(0) {}
    int numero_de_eixos() const { return _numero_eixos; }
    void muda_numero_de_eixos(int novo)
    {
        if(novo > 0) _numero_eixos = novo;
    }
    double potencia() const { return _potencia; }
    void muda_potencia(double nova)
    {
        if(nova > 0) _potencia = nova;
    }
};
```

[~]$ _

>>> Exemplo

```
#include<iostream>
int main() {
    Veiculo uno;
    uno.muda_potencia(65);
    std::cout << "Potencia por eixo: "
              << uno.potencia() / uno.numero_de_eixos()
              << std::endl;
    return 0;
}
```

[~]$ _

>>> Exemplo

```
class Veiculo_de_carga : public Veiculo {
    double _carga_maxima;
public:
    Veiculo_de_carga() : _carga_maxima(0) {}
    double carga_maxima() const { return carga_maxima; }
    void muda_carga_maxima(double nova) {
        if (nova > 0) _carga_maxima = nova;
    }
};
```

[~]$ _

>>> Exemplo

```cpp
#include<iostream>
int main() {
    Veiculo_de_carga scania;
    scania.muda_numero_de_eixos(3);
    scania.muda_potencia(730);
    scania.muda_carga_maxima(120000);

    std::cout << "Potencia por eixo: "
              << scania.potencia() / scania.numero_de_eixos()
              << std::endl;
    return 0;
}
```

[~]$ _

>>> **Exemplo**

```
#include<iostream>
class Contador {
    int _n;
public:
    Contador() : n(0) {}
    void anda() { _n++; }
    int valor() { return _n; }
};

int main() {
    Contador tranquilo;
    tranquilo.anda();
    std::cout << "valor: "
              << tranquilo.valor()
              << std::endl;
    return 0;
}
```

[~]$ _

>>> Exemplo

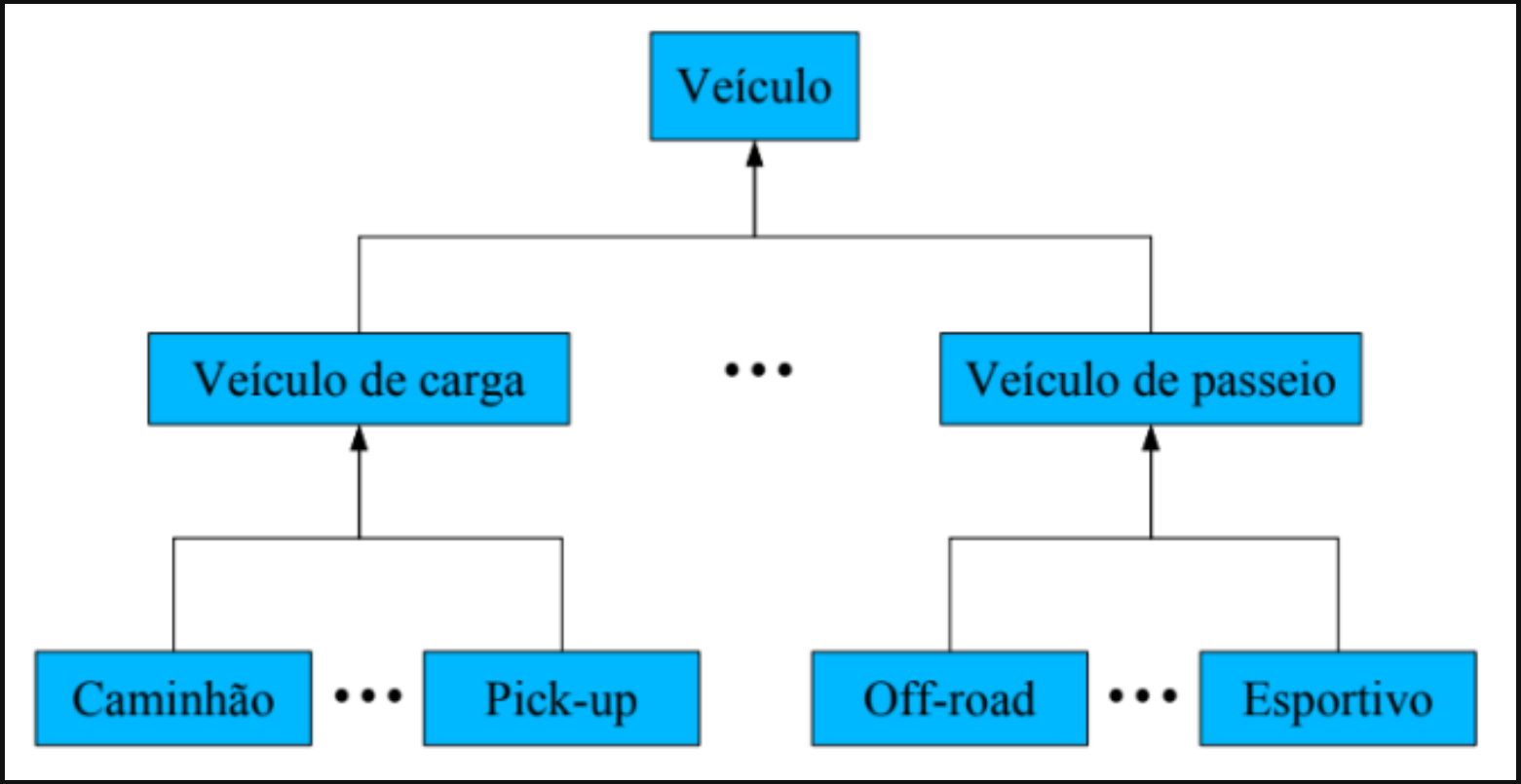
```
#include<iostream>
class Pulador : public Contador {
public:
    void anda() {
        Contador::anda();
        Contador::anda();
    }
};

int main() {
    Pulador apressado;
    apressado.anda();
    std::cout << "valor: "
              << apressado.valor()
              << std::endl;
    return 0;
}
```

[~]$ _

>>> Hierarquia de Classes

* As **relações de derivação** formam uma **hierarquia** entre as classes
* A estrutura pode não ser uma hierarquia pura, devido à existência de **herança múltipla**
* Uma classe pode **herdar de mais de uma** outra classe

>>> Hierarquia Simples




```
class Veiculo { ... };
class Veiculo_de_carga : public Veiculo { ... };
class Veiculo_de_passeio : public Veiculo { ... };
class Caminhao : public Veiculo_de_carga { ... };
class Pickup : public Veiculo_de_carga { ... };
class Off_road : public Veiculo_de_passeio { ... };
class Esportivo : public Veiculo_de_passeio { ... };
```

[~]$ _

>>> Acesso a Membros

* public => todos (inclusive classes)
* private => apenas métodos da classe e amigos (nem classe derivada)
* protected => métodos da classe e das derivadas

[~]$ _

>>> Exemplo

```
class Contador {
protected:
    int _n;
public:
    Contador() : n(0) {}
    void anda() { _n++; }
    int valor() const { return _n; }
};

class Pulador : public Contador {
public:
    void anda() {
        _n += 2;
    }
};
```

[~]$ _

>>> Cuidados

* Restrição: public < protected < private
* Vincula a implementação da classe base com todas as classes derivadas (perde o encapsulamento)
* Código difícil de alterar

>>> Tipos de Herança

* Três tipos de herança:
  * Pública (public)

    ```
    class Veiculo_de_carga : public Veiculo { ... };
    ```
  * Privada (private)
  * Protegida (protected)
* Apenas **public** = **é-um**
* **private** e **protected** -> **reaproveitamento de código**

[~]$ _

>>> Herança Pública

Herança pública: mantém a acessibilidade da classe base

* public -> public
* private -> private
* protected -> protected

```
class Base { ... };
class Derivada : public Base { ... };
```

[~]$ _

>>> Herança Privada

Herança privada: transforma todos os membros em privados (esconde a herança do cliente e das classes derivadas)

* public -> private
* private -> private
* protected -> private

```
class Base { ... };
class Derivada : private Base { ... };
```

>>> **Exemplo**

```
class Lampada {
    bool _ligada;
public:
    Lampada() : _ligada(false) {}
    void liga() { _ligada = true; }
    void desliga() { _ligada = false; }
    bool ligada() { return _ligada; }
};
class Motor : private Lampada {
    double _rpm;
public:
    void liga() { Lampada::liga(); }
    void desliga() { Lampada::desliga(); }
    double rotacao() {
        if (Lampada::ligada()) return _rpm;
        else return 0;
    }
};
```

[~]$ _

>>> Herança Protegida

Herança protegida: os membros públicos herdados passam a ser protected (classes derivadas sabem sobre a herança, mas clientes não)

* public -> protected
* private -> private
* protected -> protected

```
class Base { ... };
class Derivada : protected Base { ... };
```

>>> Acessibilidade Membro a Membro

* É possível controlar acessibilidade membro a membro
* Proibido abrir acesso a membro fechado pela classe base
* Proibido fechar acesso a membro liberado pela classe base
* Apenas para heranças private e protected

[~]$ _

>>> **Exemplo**

```
class Lampada {
    bool _ligada;
public:
    Lampada() : _ligada(false) {}
    void liga() { _ligada = true; }
    void desliga() { _ligada = false; }
    bool ligada() { return _ligada; }
};
class Motor : private Lampada {
    double _rpm;
public:
    using Lampada::liga;
    using Lampada::desliga;
    double rotacao() {
        if (Lampada::ligada()) return _rpm;
        else return 0;
    }
};
```

[~]$ _

>>> Compatibilidade de Ponteiros

* Ponteiros de tipos diferentes não podem ser misturados (Exceção: void *)
* Com herança há uma exceção
* Ponteiros da classe base podem apontar para objetos de classe derivada
* A outra direção não é permitida

[~]$ _

>>> Exemplo

```
Veiculo *pv;
Veiculo_de_carga *pvc;
Veiculo_de_passeio *pcp;
Caminhao *pc;
Off_road *po;
pc = new Caminhao();
po = new Off_road();
pvc = pc; // OK
pcp = po; // OK
pv = pvc; // OK
pv = pcp; // OK
pv = pc; // OK
pv = po; // OK
pcp = pc; // Erro
pc = pcp; // Erro
```

>>> Herança Múltipla

* É possível a uma **classe herdar** membros de **mais de uma classe base**
* => **herança múltipla**
* **Erro** se herdados **membros com mesmo nome** de classes-base distintas

[~]$ _

>>> **Exemplo**

```
class Relogio {
...
public:
    int horas() const;
    int minutos() const;
    int segundos() const;
...
};

class Patrimonio {
...
public:
    double valor() const;
...
};

class Rolex : public Relogio, public Patrimonio {
...
};
```

[~]$ _

>>> Construtores e Destruidores

* classe B é derivada da classe A
* Ao construir objeto da classe B, estamos também construindo objeto da classe A
* **Construtor** da classe **base** chamado **antes**
* Em **herança múltipla**, usa **ordem de declaração**
* **Idem** (ordem ao contrário) para **destruidores**

[~]$ _