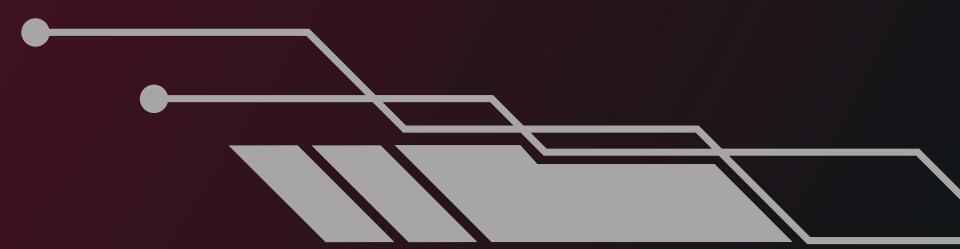




TRAINING

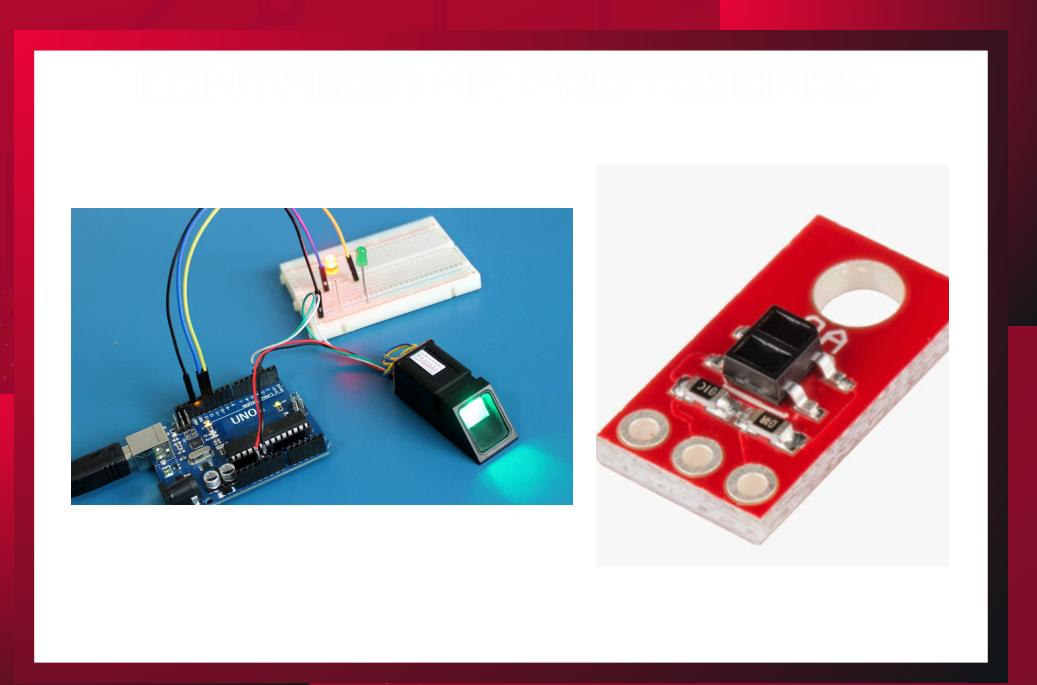




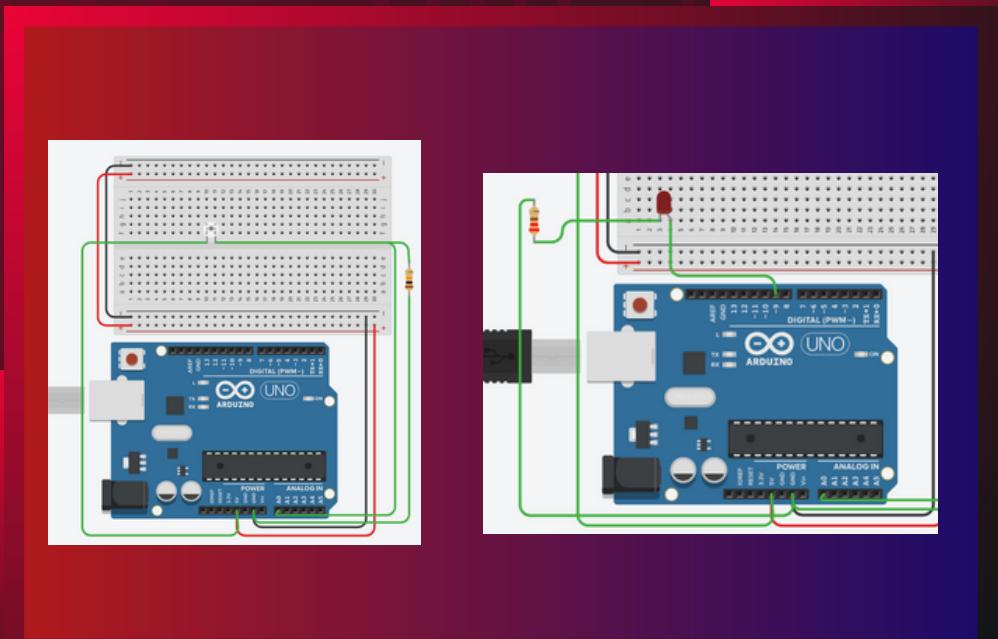
RETROSPECTIVA

- USO DE SENsoRES NO ARDUINO
- SENSOR ANALÓGICO VS DIGITAL
- COMO CONECTAR SENsoRES NO ARDUINO

SENSOR ANALÓGICO VS
SENSOR DIGITAL



COMO CONECTAR
SENsoRES NO ARDUINO

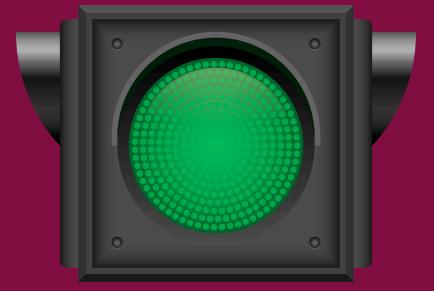
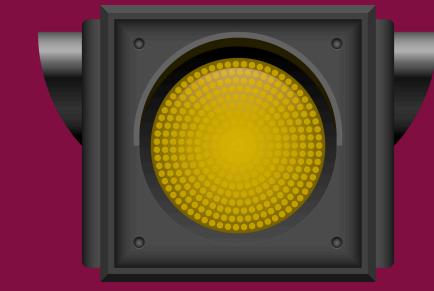
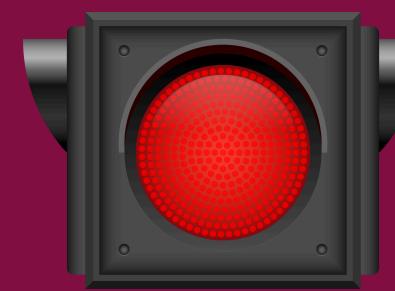


BOOTCAMP DA
ROBOCAMP



EXERCÍCIOS DE FIXAÇÃO

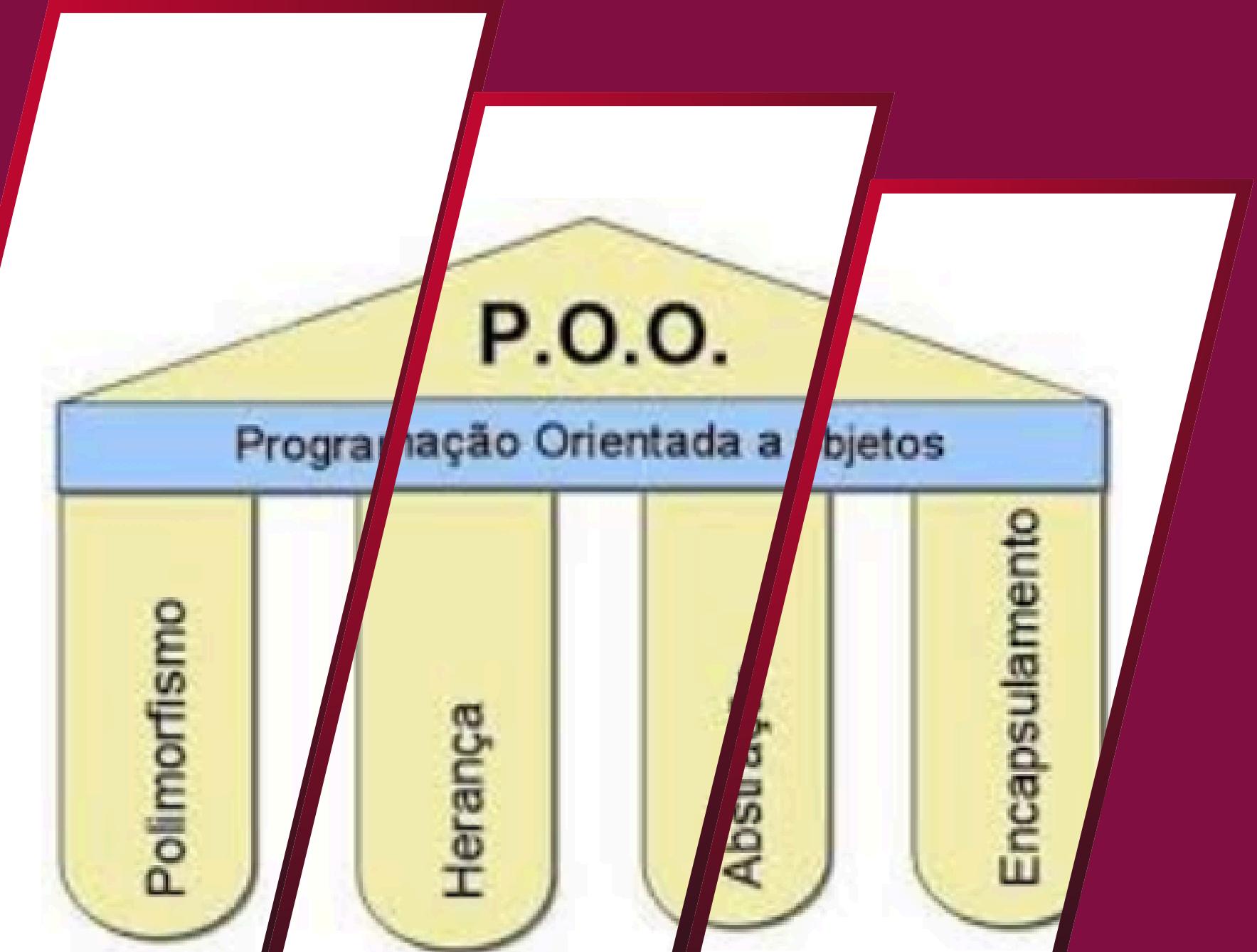
SEMÁFORO COM PEDESTRE
ORIENTADO À OBJETOS





INTRODUÇÃO A PROGRAMAÇÃO ORIENTADA A OBJETOS (POO)

- O QUE É UM PARADIGMA DE PROGRAMAÇÃO?
- ESTRUTURADO VS O.O.
- CLASSES X OBJETOS
- PILARES DA O.O.
- HERANÇA





ESSA AULA:



OBJETIVO

SOMENTE APRESENTAR A PROGRAMAÇÃO ORIENTADA A OBJETOS E OS CONCEITOS DE CLASSES E OBJETOS PARA MELHOR ESTRUTURAÇÃO DE PROJETOS, COESÃO ENTRE COMPONENTES E DIVISÃO DE RESPONSABILIDADES



O QUE É UM PARADIGMA DE PROGRAMAÇÃO?

- A FORMA MENOS ABSTRATA: É UM JEITO ORGANIZAR E ESCREVER CÓDIGO PARA RESOLVER UM PROBLEMA COM UM COMPUTADOR.
- A FORMA MAIS ABSTRATA: É A IMPOSIÇÃO DE DISCIPLINA SOBRE O ESTILO DE ESCRITA DE CÓDIGO, RESTRINGINDO CERTAS AÇÕES PARA PROMOVER ORGANIZAÇÃO, CLAREZA E MANUTENÇÃO NO DESENVOLVIMENTO DE SOFTWARE.



PARADIGMAS DE PROGRAMAÇÃO:

HOJE EM DIA EXISTEM 3
GRANDES PARADIGMAS DE
PROGRAMAÇÃO:

- PARADIGMA ESTRUTURADO
- PARADIGMA FUNCIONAL
- PARADIGMA ORIENTADO A OBJETOS



PARADIGMA ESTRUTURADO: EXEMPLO

CONSIDERE A SEGUINTE
ESTRUTURA EM C:

```
typedef struct carro {  
    char *marca;  
    char *modelo;  
    char *placa;  
    int ano;  
} Carro;
```



PARADIGMA ESTRUTURADO: EXEMPLO

CONSIDERE A SEGUINTE
ESTRUTURA EM C:

```
typedef struct carro {  
    char *marca;  
    char *modelo;  
    char *placa;  
    int ano;  
} Carro;
```

EM UM DIAGRAMA SIMPLES,
REPRESENTEMOS ELA DA
SEGUINTE FORMA:

«struct»
Carro

```
Char *marca;  
Char *modelo;  
Char *placa;  
int ano;
```



PARADIGMA ESTRUTURADO: EXEMPLO

```
typedef struct carro {  
    char *marca;  
    char *modelo;  
    char *placa;  
    int ano;  
} Carro;
```

Carro *newCarro(args...) - Função construtora.

void carro_andarKm(Carro *carro, int km) - Função que atua sob/sobre a estrutura Carro.

NA PROGRAMAÇÃO ESTRUTURADA, USAMOS FUNÇÕES PARA MANIPULAR E ALTERAR O ESTADO DAS NOSSAS ESTRUTURAS.

```
Carro *newCarro(char *marca, char *modelo, char *placa, int ano) {  
    Carro *novo_carro = (Carro *)malloc(sizeof(Carro));  
    if (!novo_carro)  
        return NULL;  
  
    novo_carro->marca = marca;  
    novo_carro->modelo = modelo;  
    novo_carro->placa = placa;  
    novo_carro->ano = ano;  
  
    return novo_carro;  
}  
  
void carro_andarKm(Carro *carro, int km) {  
    printf("O %s %s com placa %s rodou %d kilometros...\n", carro->marca  
          | carro->modelo, carro->placa, km);  
}
```



PARADIGMA ESTRUTURADO: EXEMPLO

EXECUTANDO NOSSA ESTRUTURA COM A
SEGUINTE MAIN SIMPLES:

```
int main(void) {
    Carro *carro = newCarro("Hyundai", "HB20", "R0B0-CAMP", 2022);

    carro_andarKm(carro, 6000);

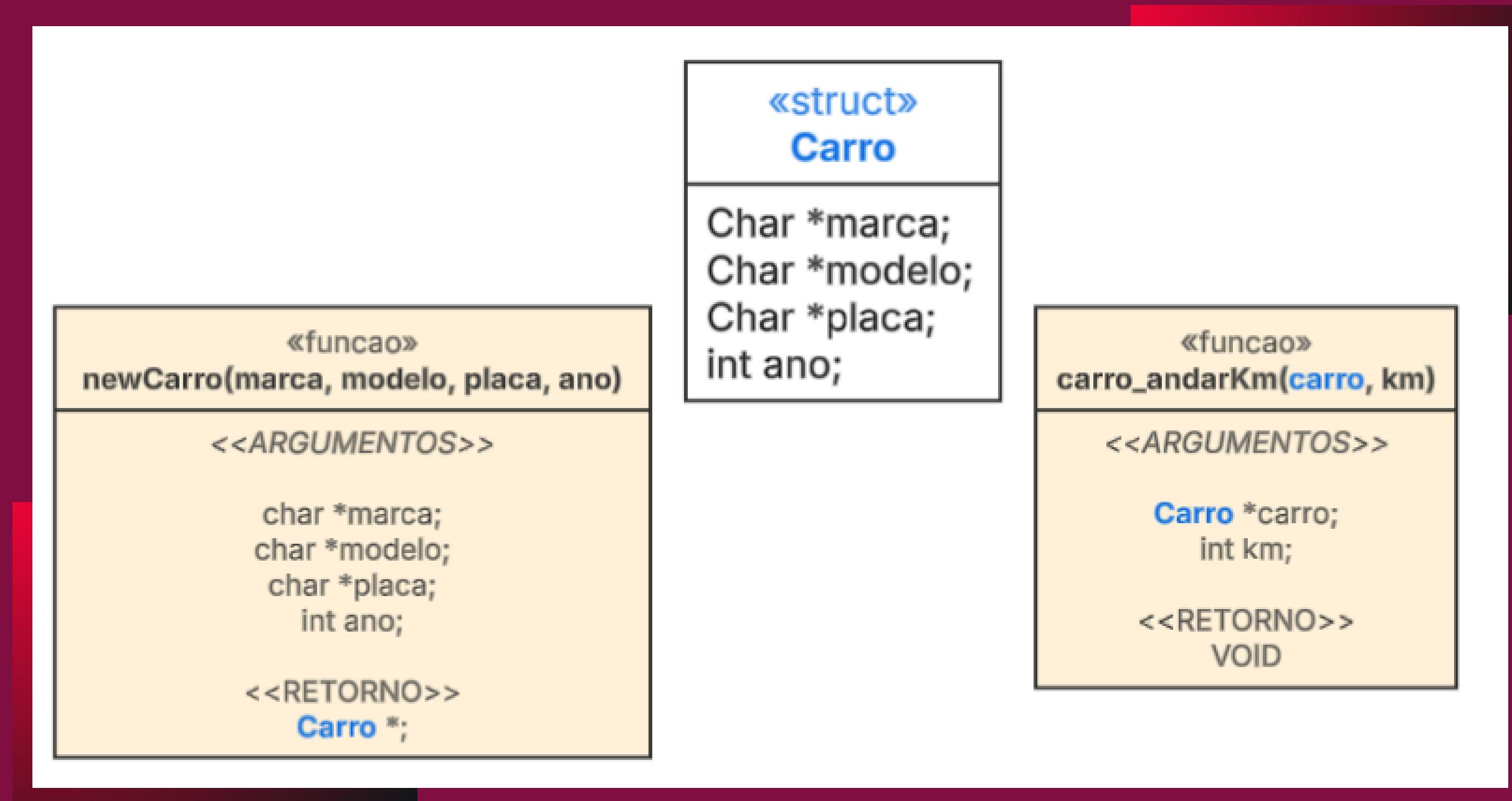
    return 0;
}

A ~/Documents/dmp
./main
0 Hyundai HB20 com placa R0B0-CAMP rodou 6000 kilometros ...
```



PARADIGMA ESTRUTURADO: EXEMPLO

TRAZENDO A ESTRUTURA E AS FUNÇÕES EM UM DIAGRAMA,
TEMOS 3 ELEMENTOS SEPARADOS QUE COOPERAM APENAS
POR CONVENÇÃO E DISCIPLINA (DO PROGRAMADOR):



PROGRAMAÇÃO ORIENTADA A OBJETOS

NA OO. PODEMOS DEFINIR ATRIBUTOS E COMPORTAMENTOS DE NOSSAS ESTRUTURAS E ASSOCIAR TUDO DE UMA FORMA MAIS COESA E CONSISTENTE DO QUE NA PROGRAMAÇÃO ESTRUTURADA.

CHAMAMOS DE CLASSES, ESSAS ESTRUTURAS QUE DEFINEM CAMPOS DE ATRIBUTOS E COMPORTAMENTOS:

- CHAMAMOS DE ATRIBUTOS OS CAMPOS QUE GUARDAM VALORES CONEXOS (COMO EM ESTRUTURAS)
- CHAMAMOS DE MÉTODOS AS FUNÇÕES QUE ASSOCIAMOS AS CLASSES/ OBJETOS.

VEJAM: UNIFIED MODELING LANGUAGE (UML)

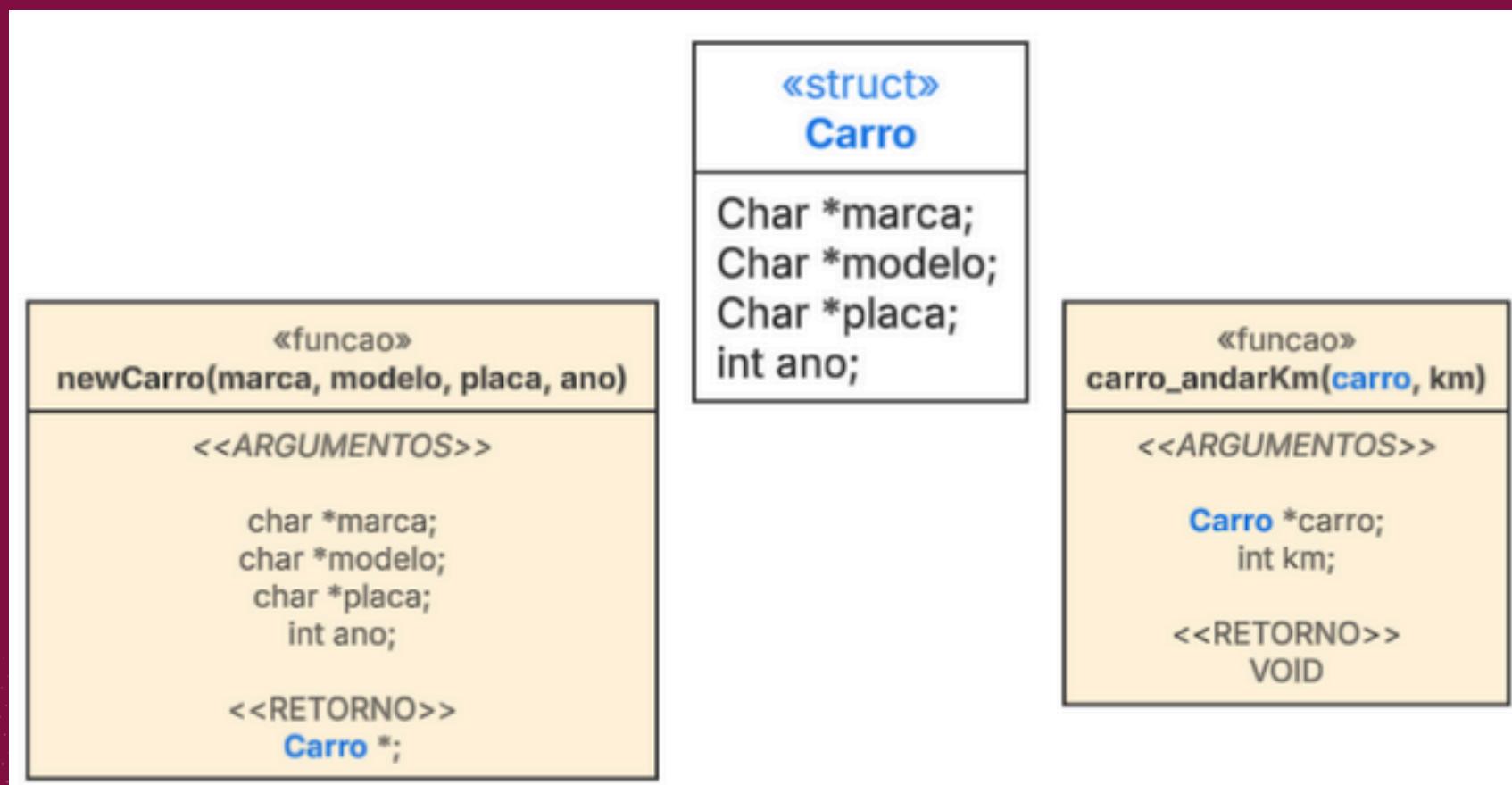
Classe
(+/-/#) tipo : nomeDoAtributo
(+/-/#) retorno nomeDoMétodo(parâmetros);



PROGRAMAÇÃO ORIENTADA A OBJETOS

Enquanto antes (em programação estruturada) tínhamos elementos separados, na Orientação a Objetos conseguimos agrupar os procedimentos de uma estrutura de uma forma mais coesa e simplificada:

Paradigma Estruturado



Paradigma Orientado a Objetos

PARADIGMA OO: EXEMPLO

AO TRAZER A IMPLEMENTAÇÃO ANTERIOR PARA CÓDIGO ORIENTADO A OBJETOS (C++), PODEMOS TER A SEGUINTE IMPLEMENTAÇÃO:

```
class Carro {  
public:  
    std::string marca;  
    std::string modelo;  
    std::string placa;  
    int ano;  
  
    Carro(std::string _marca, std::string _modelo, std::string _placa, int _ano) {  
        this->marca = _marca;  
        this->modelo = _modelo;  
        this->placa = _placa;  
        this->ano = _ano;  
    }  
  
    void andarKm(int km) {  
        std::cout << "O " << this->marca << " " << this->modelo << " com placa "  
             << this->placa << " rodou " << km << " kilometros\n";  
    }  
};
```



PARADIGMA OO: EXEMPLO

Perceba que, dessa vez, a instância de Carro (variável carro) que faz a chamada da função andarKm().

O programador tem um código mais coeso, ao acessar os componentes do carro, temos acesso a SEUS métodos e atributos.

EXECUTANDO NOSSA ESTRUTURA COM A SEGUINTE MAIN SIMPLES:

```
int main(void) {  
    Carro *carro = new Carro("Hyundai", "HB20", "R0BO-CAMP", 2022);  
  
    carro->andarKm(6000);  
  
    return 0;  
}
```

```
▲ ~/Documents/dmp  
./main
```

O Hyundai HB20 com placa R0BO-CAMP rodou 6000 kilómetros

A IDE (LSP) normalmente sugere no autocomplete os elementos acessíveis que o objeto possui.

carro->

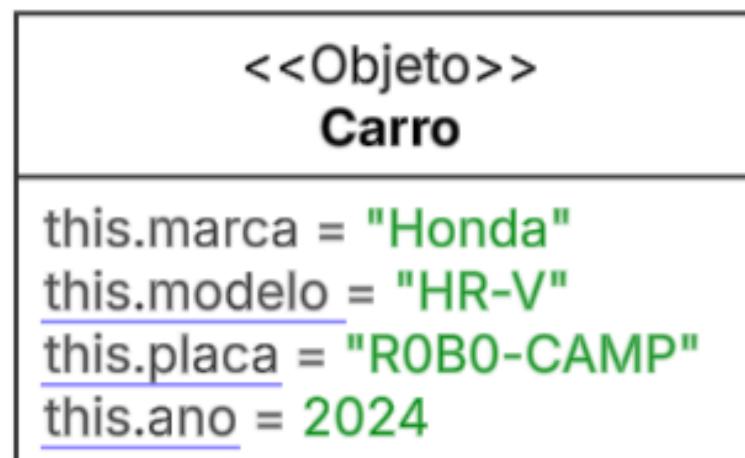
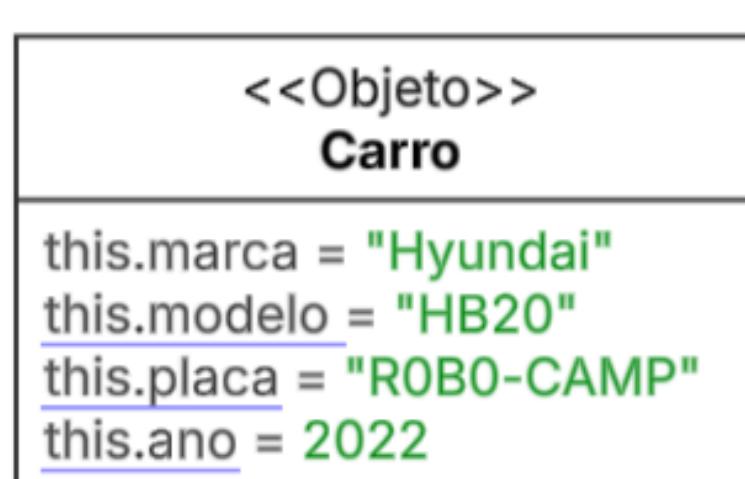
- ↳ andarKm~ (int km)
- ↳ ano
- ↳ marca
- ↳ modelo
- ↳ placa



CLASSE VS OBJETO

PODEMOS FAZER A SEGUINTE DISTINÇÃO
ENTRE UMA CLASSE E UM OBJETO:

Classes e Objetos



Uma classe define um contrato de comportamentos e atributos que suas instâncias (seus objetos) irão possuir. Uma classe está para um objeto quase como uma receita está para um produto.



CÓDIGO 0.0. EM C++:

Declaração do nome da classe, e abertura de seu escopo.

Bloco de declaração de nível de acesso. Em c++ pode ser Público (public), Privado (private) ou Protegido (Protected). Para nossa aula de hoje, todas as declarações serão públicas. Isso é chamado de Encapsulamento.

Declaração do método construtor da classe.

Aqui, criamos uma variável ponteiro e passamo uma nova instância (objeto) de Carro. Para sua criação, invocamos o método construtor declarado anteriormente.

```
1 #include <iostream>
1 #include <string>
2
3 class Carro {
4 public:
5     std::string marca;
6     std::string modelo;
7     std::string placa;
8     int ano;
9
10    Carro(std::string marca, std::string modelo, std::string placa, int ano) {
11        this->marca = marca;
12        this->modelo = modelo;
13        this->placa = placa;
14        this->ano = ano;
15    }
16
17    void andarKm(int km) {
18        std::cout << "O " << this->marca << " " << this->modelo << " com placa "
19        << this->placa << " rodou " << km << " kilometros\n";
20    }
21}
22
23 int main(void) {
24     Carro *carro = new Carro("Hyundai", "HB20", "R0BO-CAMP", 2022);
25
26     carro->andarKm(6000);
27
28     return 0;
29 }
```



COMPOSIÇÃO:

A ORIENTAÇÃO A OBJETOS ESTABELECE QUE NOSSOS PROGRAMAS DEVEM SER ORIENTADOS A COMPORTAMENTOS E TROCA DE MENSAGENS ENTRE CLASSES E OBJETOS.

RECOMENDA-SE CRIAR NOSSAS CLASSES COMO TIPOS DE DADOS COM MAIOR ÊNFASE EM SEU ESCOPO COMPORTAMENTAL.

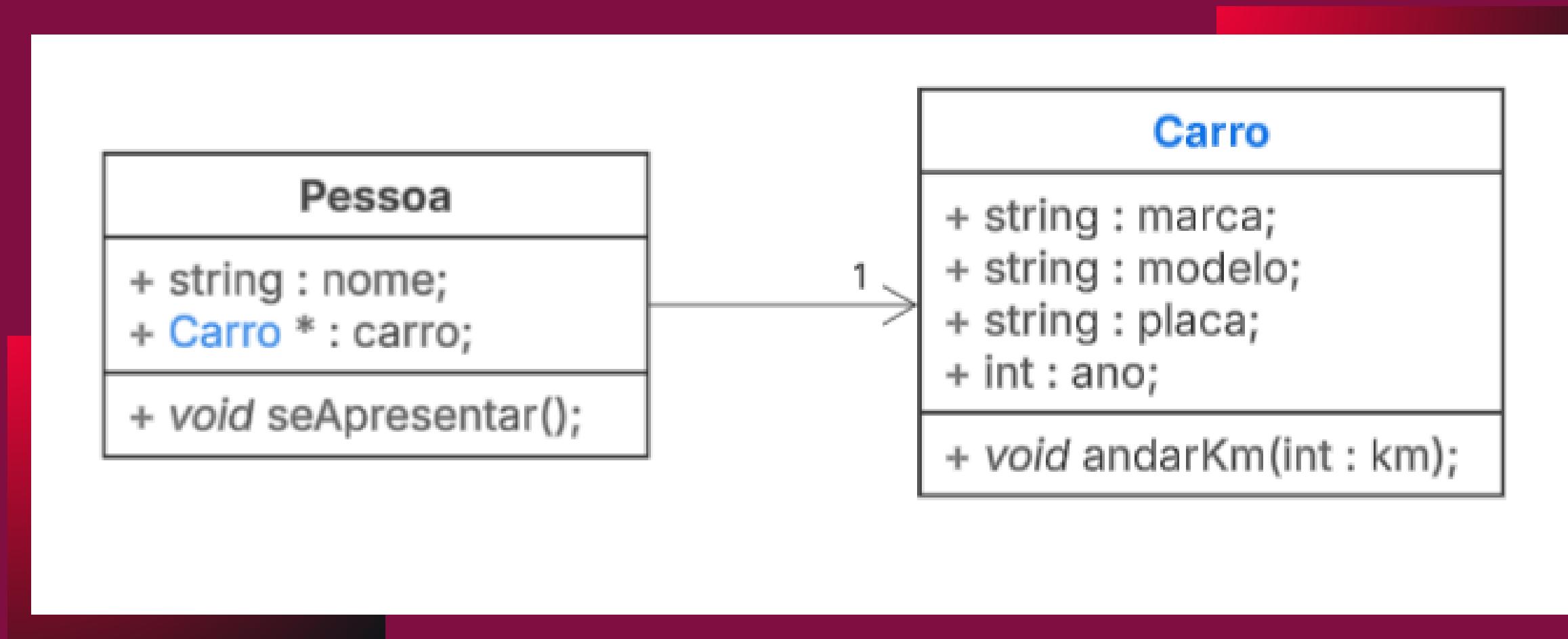
FAZEMOS DIZERAS PERGUNTAS PARA O ORIENTAR NOSSA CRIAÇÃO DE CLASSES:

- O QUE PRECISO PARA MEU SISTEMA?
- QUAIS AS RESPONSABILIDADES DESSA CLASSE QUE ESTOU CRIANDO?
- CONSIGO DIVIDI-LA EM PARTES MENORES E, AINDA SIM, COESAS?



COMPOSIÇÃO:

EXPANDINDO UM POUCO O NOSSO EXEMPLO, PODERIA CRIAR UMA CLASSE PESSOA E ASSOCIAR UM CARRO A ELA COMO NO DIAGRAMA A SEGUIR:



Aqui, podemos ler como: Uma pessoa possui (associa) um carro. Porém, um carro não carrega referência para Pessoa.

COMPOSIÇÃO:

NO CÓDIGO, ADICIONAMOS APENAS UM CAMPO QUE REFERENCIA UM CARRO NO ATRIBUTO DA CLASSE PESSOA.

Aqui, no momento de criação da classe Pessoa, passamos uma instância (objeto) de carro como argumento. Esse carro será associado a Pessoa como um de seus atributos.

```
class Pessoa {  
public:  
    std::string nome;  
    Carro *carro;  
  
    Pessoa(std::string _nome, Carro *_carro) {  
        this->nome = _nome;  
        this->carro = _carro;  
    }  
  
    void seApresentar() {  
        std::cout << "Olá, meu nome é " << this->nome << " e tenho um "  
        << this->carro->marca << " " << this->carro->modelo << ".\n";  
    }  
};
```



COMPOSIÇÃO:

SEGUE A EXECUÇÃO DE UM CÓDIGO DEMONSTRANDO UMA INSTÂNCIA DE PESSOA QUE TEM UMA INSTÂNCIA DE CARRO ASSOCIADO.

Temos que Pessoa é uma classe que tem composição com a classe Carro.

```
int main(void) {
    Carro *carro = new Carro("Hyundai", "HB20", "RÔBO-CAMP", 2022);

    Pessoa *pedro = new Pessoa("Pedro", carro);
    pedro->seApresentar();

    return 0;
}
```

```
▲ ~ /Documents/dmp
./main
```

Olá, meu nome é Pedro e ~~Não~~ tenho um Hyundai HB20.



HERANÇA:

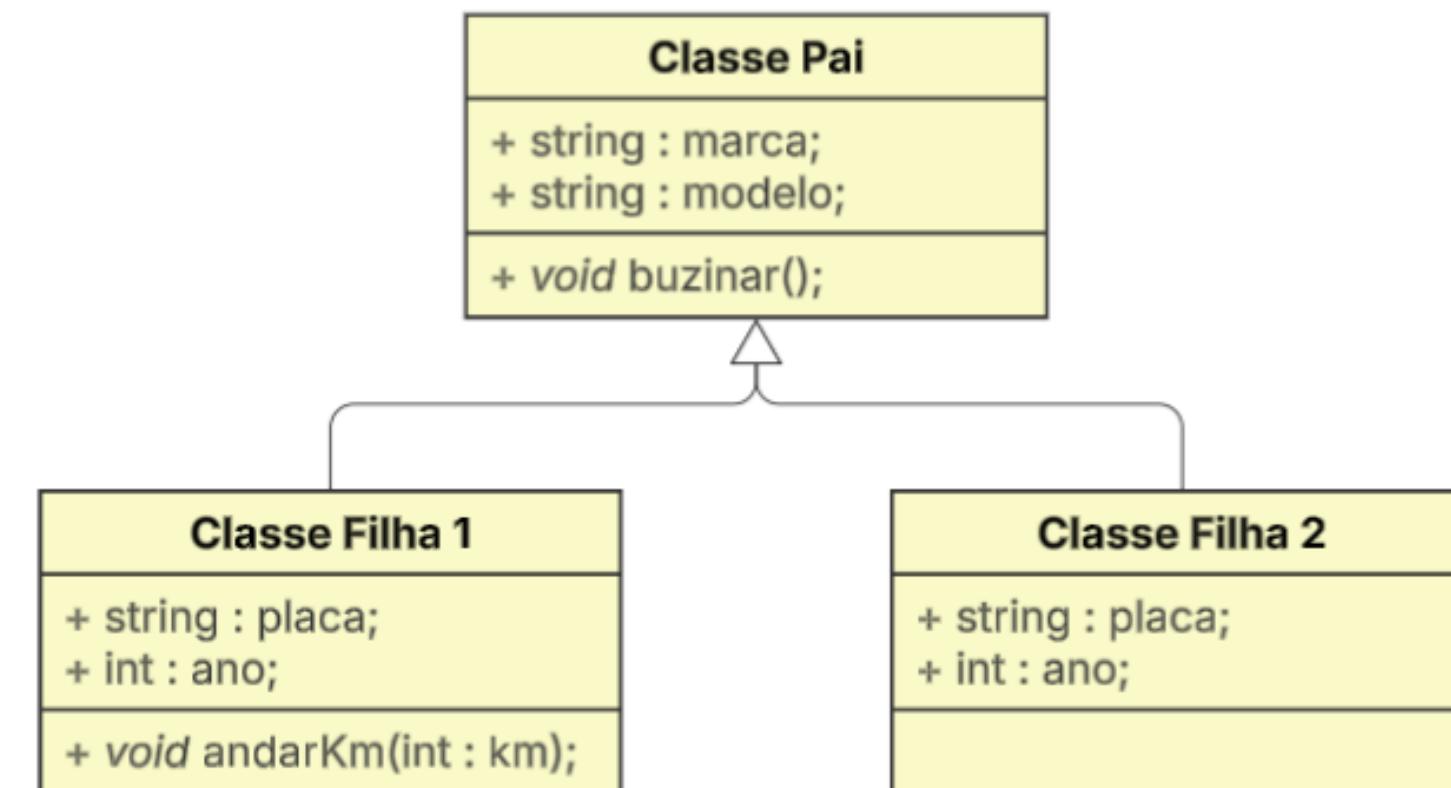
UMA CLASSE PODE HERDAR DE UM OUTRA CLASSE E, DESSA FORMA, HERDAR TODOS OS MÉTODOS E ATRIBUTOS PÚBLICOS E PRIVADOS DE SUA CLASSE PAI.

Quando uma classe é especializada, dizemos que ela é um membro da classe pai.

Nesse caso, “Classe Filha 1 é uma Classe Pai”.

Fará mais sentido a seguir.

Herança Simples

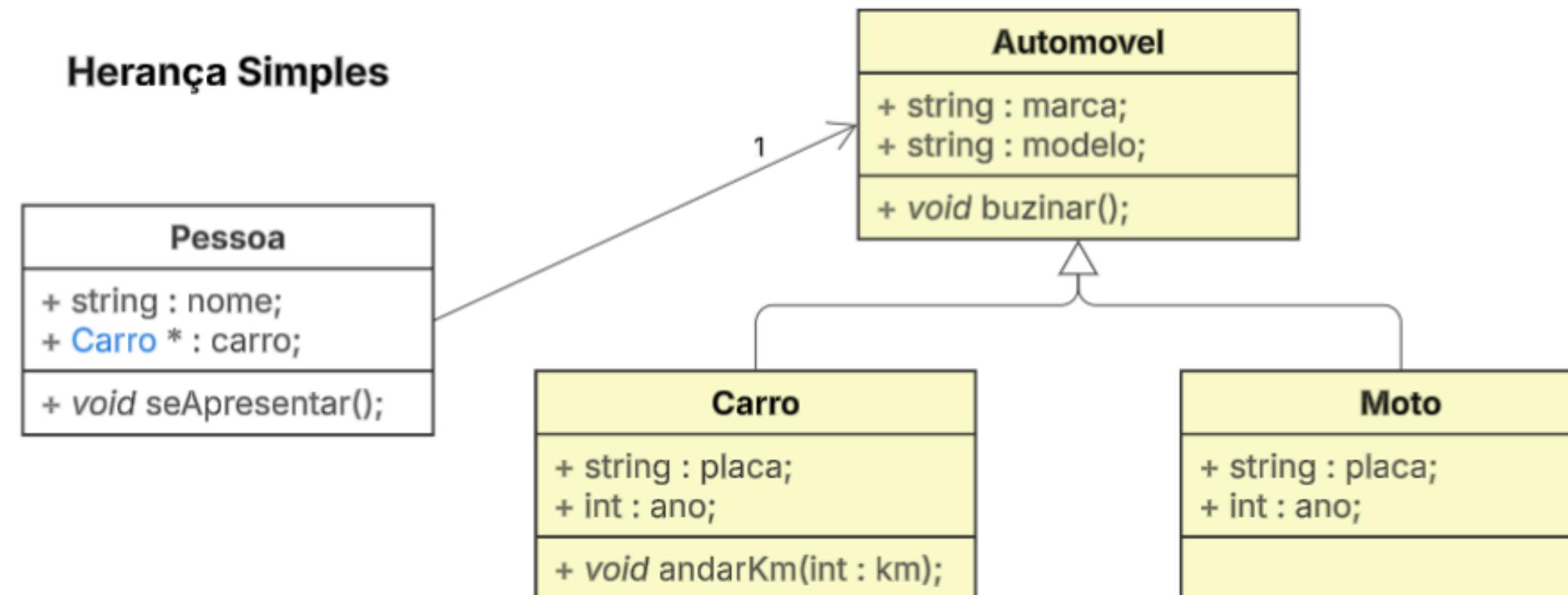


HERANÇA:

NO NOSO EXEMPLO, PODEMOS GENERALIZAR ALGUNS ATRIBUTOS PARA UMA CLASSE CHAMADA AUTOMOVEL E CRIAR DUAS SUBCLASSES CHAMADO CARRO E MOTO.

Perceba que, assim, podemos fazer com que Pessoa seja associada a um Automovel, pois os atributos de carro que invocavamos nela foram generalizados.

Herança Simples



HERANÇA EM C++:

Em C++, a herança é declarada usando o operador “.”. Somente os membros public e protected da classe pai são herdados.

Como dito anteriormente, nesse caso, dizemos que “Carro é um Automovel” e “Moto é um Automovel”.

No método construtor de uma classe filha, invocamos o método construtor de sua classe pai, para garantir a inicialização adequada dos campos.

```
class Automovel {  
public:  
    std::string marca;  
    std::string modelo;  
  
    Automovel(std::string marca, std::string modelo) { ...  
    }  
  
    void buzinar() { std::cout << "Biiiii!\n"; }  
};  
  
class Carro : public Automovel {  
public:  
    std::string placa;  
    int ano;  
  
    Carro(std::string _marca, std::string _modelo, std::string _placa, int _ano)  
        : Automovel(_marca, _modelo) {  
            this->placa = _placa;  
            this->ano = _ano;  
    }  
  
    void andarKm(int km) { ...  
    }  
};  
  
class Moto : public Automovel {  
public:  
    std::string placa;  
    int ano;  
  
    Moto(std::string _marca, std::string _modelo, std::string _placa, int _ano)  
        : Automovel(_marca, _modelo) {  
            this->placa = _placa;  
            this->ano = _ano;  
    }  
};
```



HERANÇA EM C++:

Dizer que um Carro é um Automóvel é mais literal do que figurativo em C++.

Ao requisitar uma classe pai, podemos passar uma classe filha em seu lugar e o funcionamento do programa deve ser garantido.

Especialização não deve “quebrar” a coesão do código. Faz mais sentido se observarmos a função main(void).

Perceba que a classe Pessoa sofreu poucas alterações para adaptar o Carro para sua classe pai.

A execução desse código é exatamente igual ao exemplo anterior.

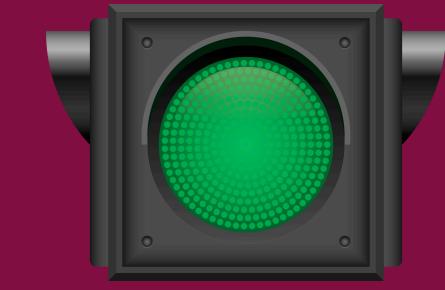
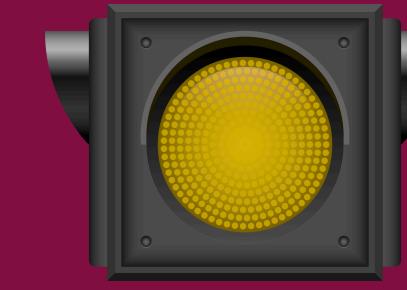
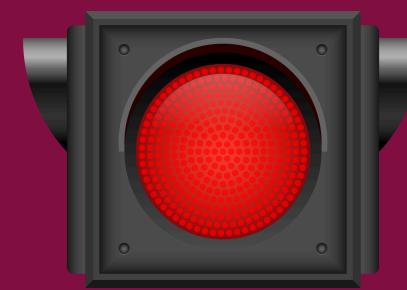
Mas veja que passamos como argumento uma classe especializada (Carro) onde era requisitado uma classe generalizada (Automovel).

```
class Pessoa {  
public:  
    std::string nome;  
    Automovel *automovel;  
  
    Pessoa(std::string _nome, Automovel *_automovel){  
        this->nome = _nome;  
        this->automovel = _automovel;  
    }  
  
    void seApresentar(){  
        std::cout << "Olá, meu nome é " << this->nome << " e tenho um "  
        << this->automovel->marca << " " << this->automovel->modelo  
        << ".\n";  
    }  
};  
  
int main(void){  
    Carro *carro = new Carro("Hyundai", "HB20", "ROBO-CAMP", 2022);  
  
    Pessoa *pedro = new Pessoa("Pedro", carro);  
    pedro->seApresentar();  
  
    return 0;  
}
```



EXERCÍCIO DE FIXAÇÃO

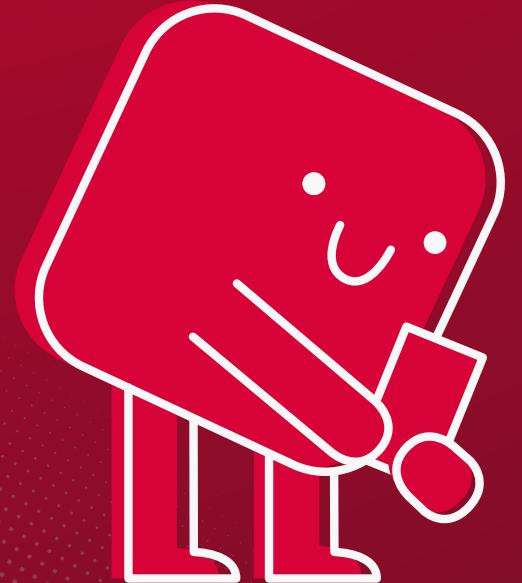
SEMÁFORO COM PEDESTRE
ORIENTADO À OBJETOS



EQUIPE RESPONSÁVEL :

PRODUÇÃO DO MATERIAL :
PEDRO EVANDRO MARTINS

DIRETOR DE TREINAMENTO :
PEDRO S. CONCEIÇÃO



MUITO
OBRIGADO !!