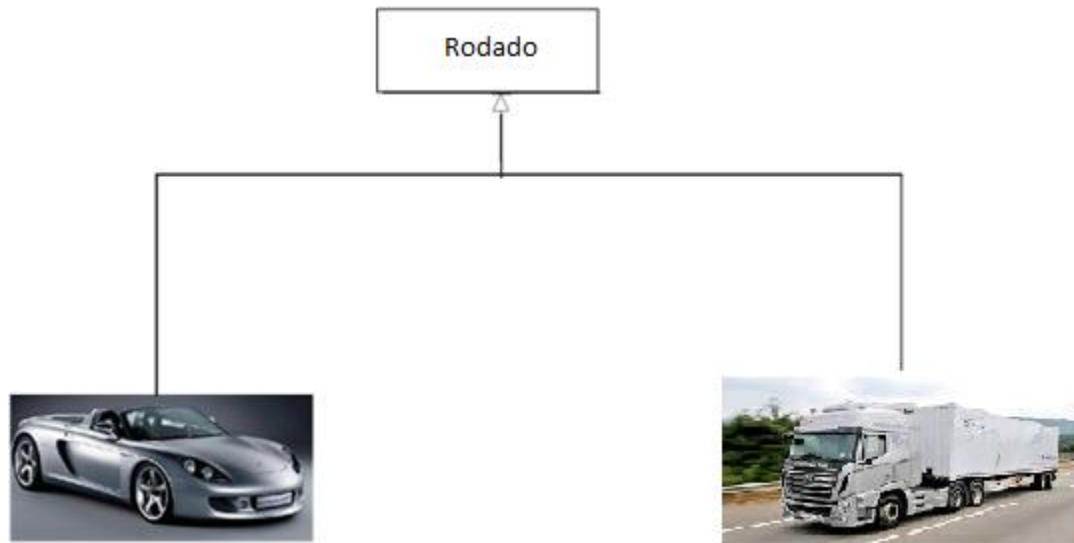

C++ HERENCIA

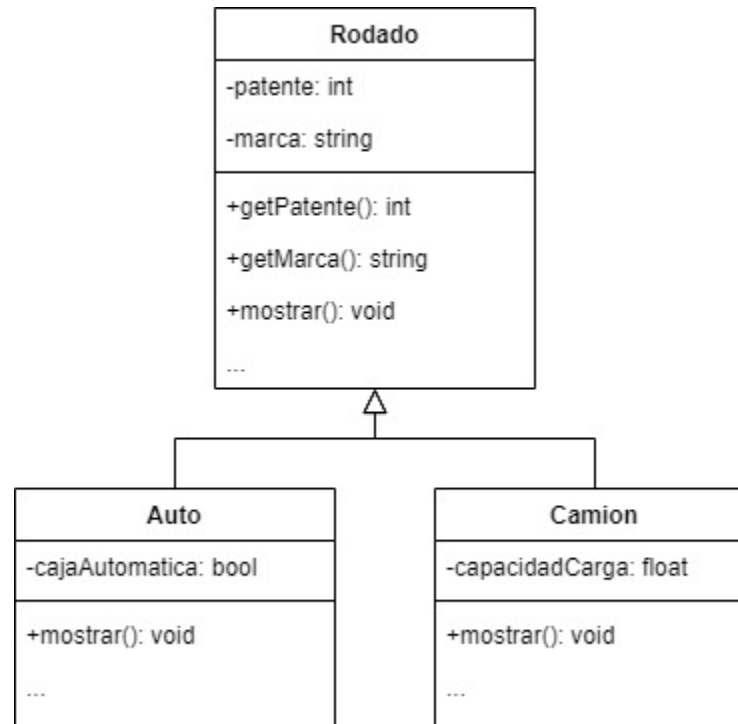
POO: principales conceptos

- Herencia



POO: principales conceptos

- Herencia



Herencia

Clase de la cual hereda	Clase que hereda
Superclase	Subclase
Padre	Hijo
Antecesor	Sucesor
Ancestro	Descendiente
Clase superior	Especialización
Clase base	Clase derivada

Herencia

- La forma general para indicar herencia es:


```
class ClaseDerivada : acceso ClaseBase
{
    //cuerpo de la nueva clase
}
```

Herencia

- Ejemplo:

```
10 // Clase Rodado
11 class Rodado
12 {
13     // Atributos
14     private:
15         string patente;
16         string marca;
17
18     // Metodos
19     public:
20         Rodado(string p, string m); // Constructor
21         string getPatente(); // Devuelve la patente
22         string getMarca(); // Devuelve la marca
23         virtual void mostrar(); // Muestra
24         virtual ~Rodado();
25 };
26
27 // Fin Clase Rodado
28
29
30 //
31 // Clase Auto
32 class Auto : public Rodado
33 {
34     // Atributos
35     private:
36         bool cajaAutomatica;
37
38     // Metodos
```

La clase Auto hereda de Rodado



Herencia

Visibilidad del miembro en la clase-base	Modificador de acceso utilizado en la declaración de la clase derivada		
	public	protected	private
public	public	protected	private (accesible)
protected	protected	protected	private (accesible)
private	private (no accesible directamente)	private (no accesible directamente)	private (no accesible directamente)

Herencia

La construcción en la herencia

- El proceso de *construcción* se realiza construyendo, en primer lugar la parte de los objetos ancestros. Luego se construye la parte correspondiente a las subclases. Es decir, si B hereda de A, en primer lugar se construye la parte de A y luego la de B.

La destrucción en la herencia

- En la *destrucción* es proceso es inverso: en primer lugar se destruye lo que corresponde a los herederos y luego se va “subiendo” y destruyendo la parte correspondiente a los ancestros.

Herencia

La construcción en la herencia

- Inicializadores.
 - Si B hereda de A, debemos llamar al constructor necesario antes que realizar cualquier otra tarea. Es por eso que se utilizan los inicializadores.

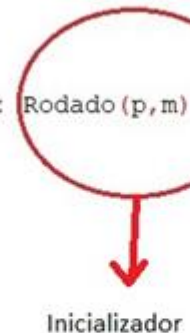
```
constructor-derivada(lista-argumentos) : base(lista-argumentos)
{
    cuerpo del constructor derivado
}
```

Herencia

La construcción en la herencia

- Inicializadores

```
35     private:
36         bool cajaAutomatica;
37
38         // Metodos
39     public:
40         // Constructor
41         Auto(string p, string m, bool caja) : Rodado(p,m)
42         {
43             cajaAutomatica = caja;
44         };
45
46         // Muestra
47         void mostrar();
48
49         // Metodo solo de auto
50         void mAuto();
51     };
52     // -----
53     // Fin Clase Auto
54     // -----
```



Inicializador

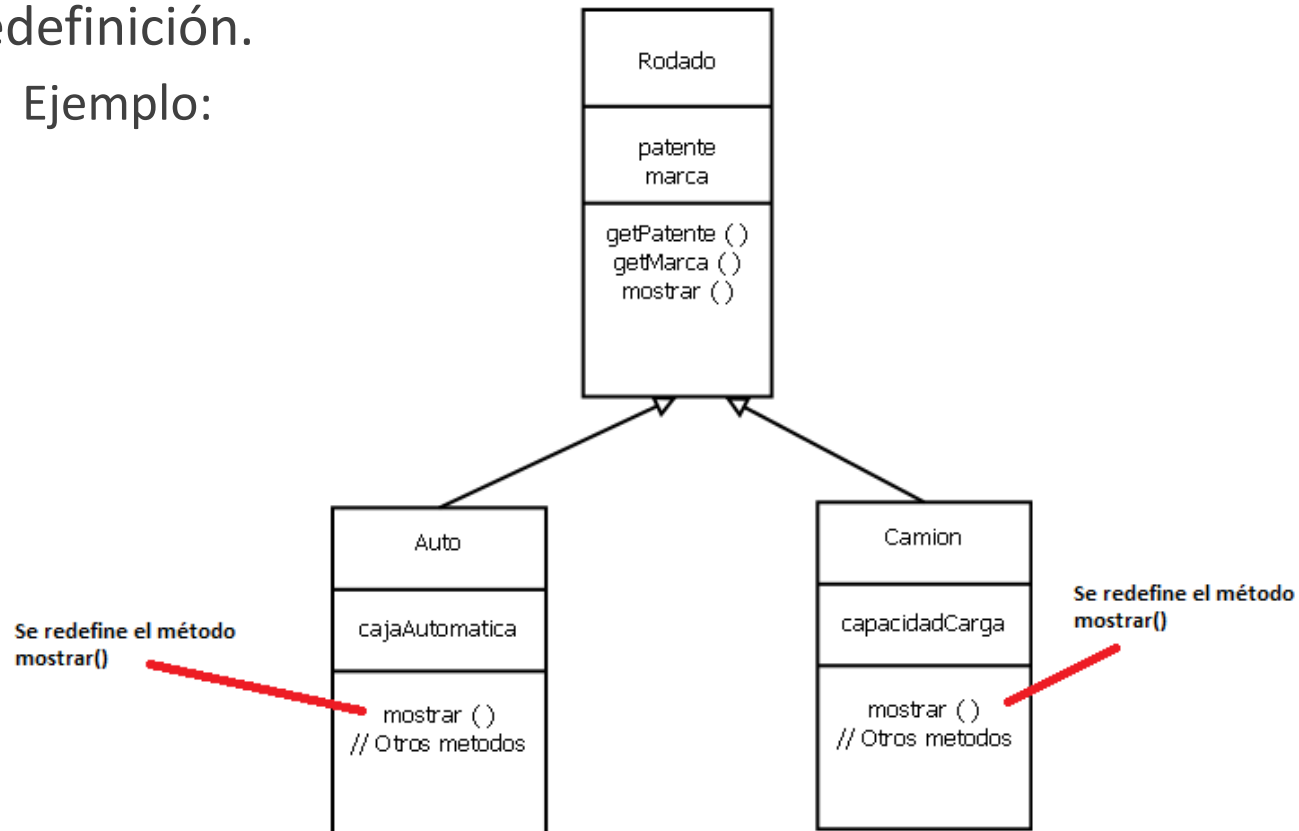
Herencia

Sobrecarga y Redefinición

- Sobrecarga.
 - Método con el mismo nombre pero difiere en sus parámetros (cantidad, tipo o ambas cosas).
- Redefinición.
 - En la herencia redefino un método (mismos parámetros).

Herencia

- Redefinición.
 - Ejemplo:



C++ POLIMORFISMO

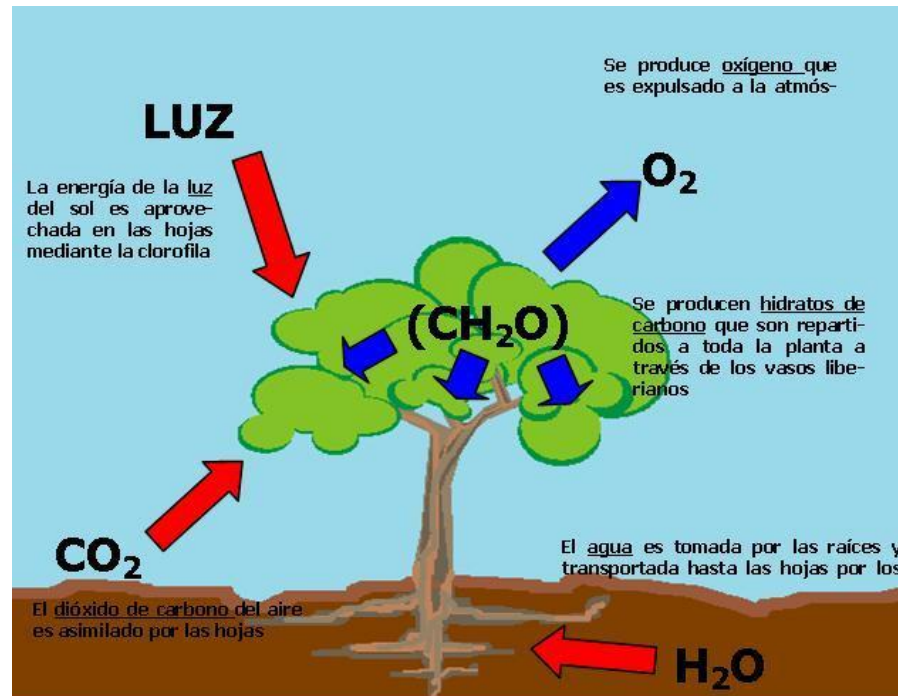
Polimorfismo

- Reacción de distintos objetos ante un mismo mensaje.
 - Aliméntate:



Polimorfismo

- Reacción de distintos objetos ante un mismo mensaje.
- Aliméntate:



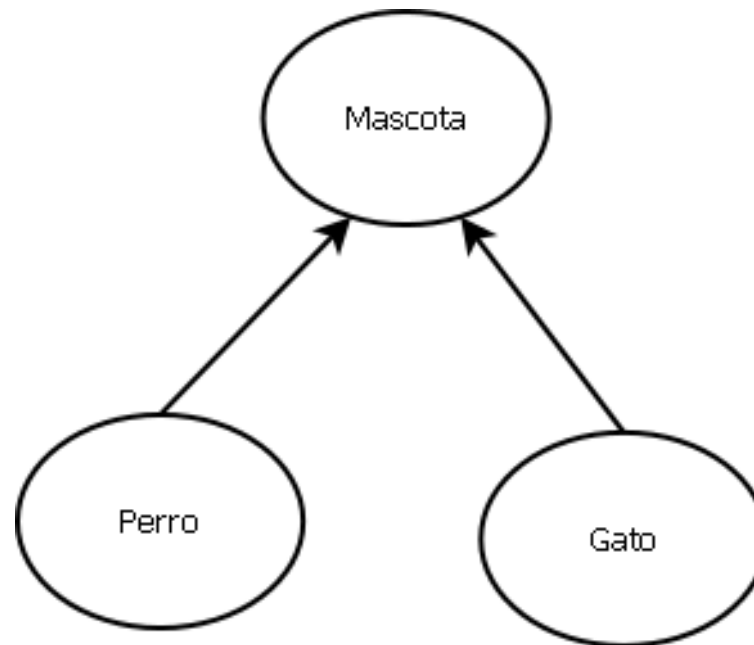
Polimorfismo

- Reacción de distintos objetos ante un mismo mensaje.
 - Aliméntate:



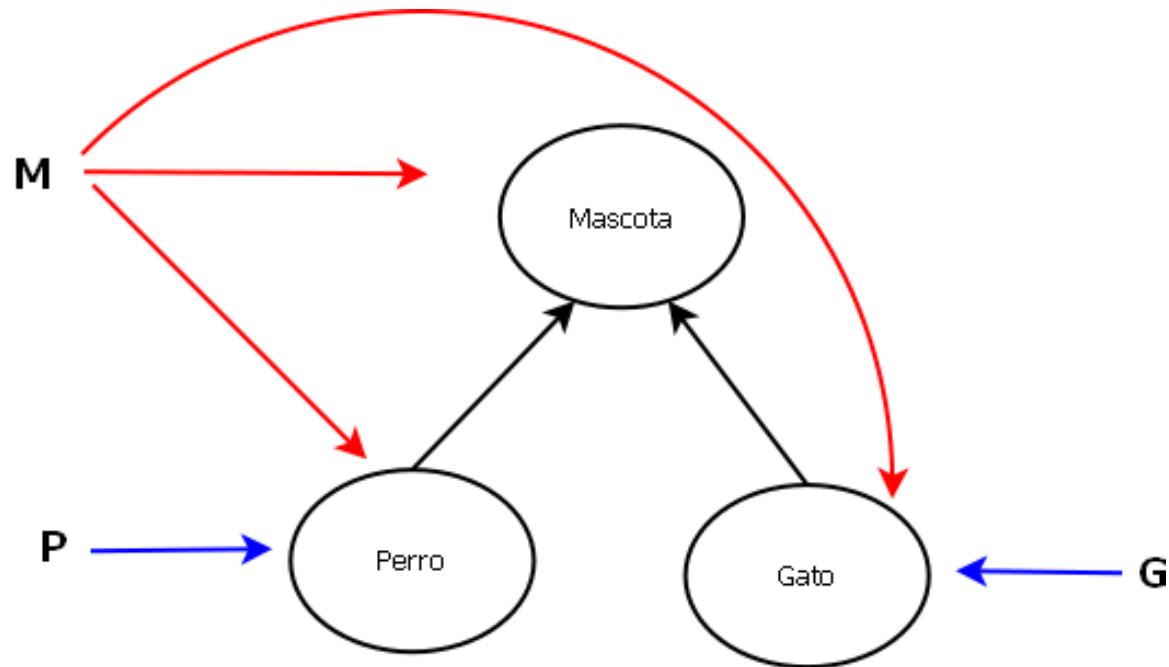
Polimorfismo

- Conexiones polimórficas:



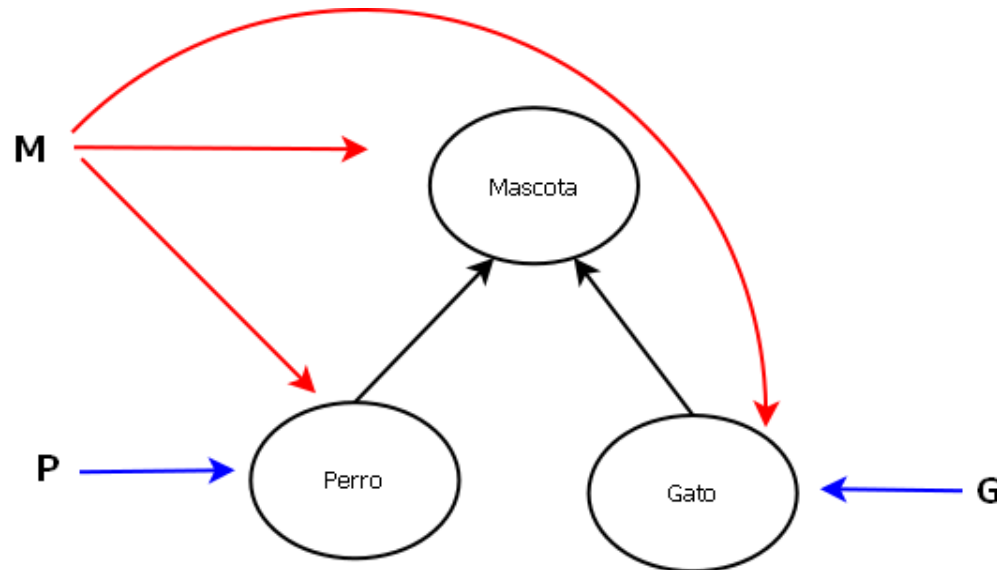
Polimorfismo

- Conexiones polimórficas:



Polimorfismo

- Conexiones polimórficas:



Un puntero de una clase ancestro sirve para apuntar a las clases herederas también, pero solo puede acceder a los atributos y métodos que tiene la clase padre.

Los punteros de las clases herederas no pueden apuntar a una clase ancestro.

Polimorfismo

- Conexiones polimórficas:

M: ref_mascota

P: ref_perro

G: ref_gato

- M := P // válido
- M := G // válido

M es un puntero a una clase de tipo Mascota.

P es un puntero a una clase de tipo Perro.

G es un puntero a una clase de tipo Gato.

Polimorfismo

- Restricciones:

M: ref_mascota

P: ref_perro

- M := objeto_perro
- P := M // Caso 1 **inválido**
- M.ladRAR () // Caso 2 **inválido**

Polimorfismo

- Restricciones:

M: ref_mascota

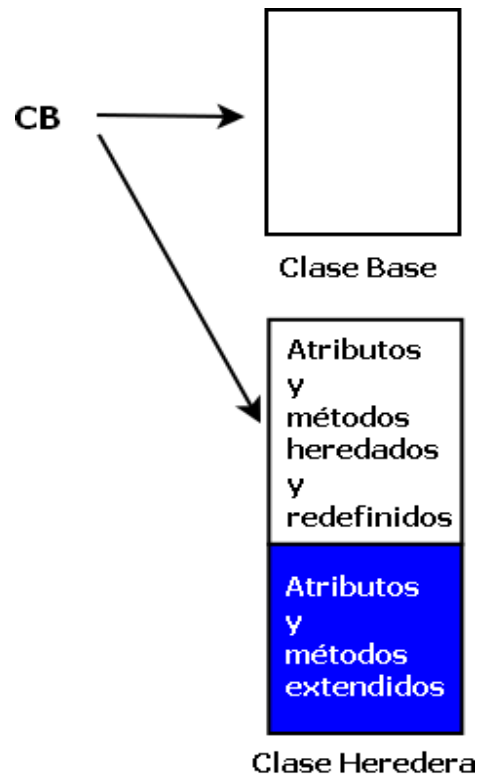
P: ref_perro

- M := objeto_perro
- P := M

// Caso 1 **inválido**

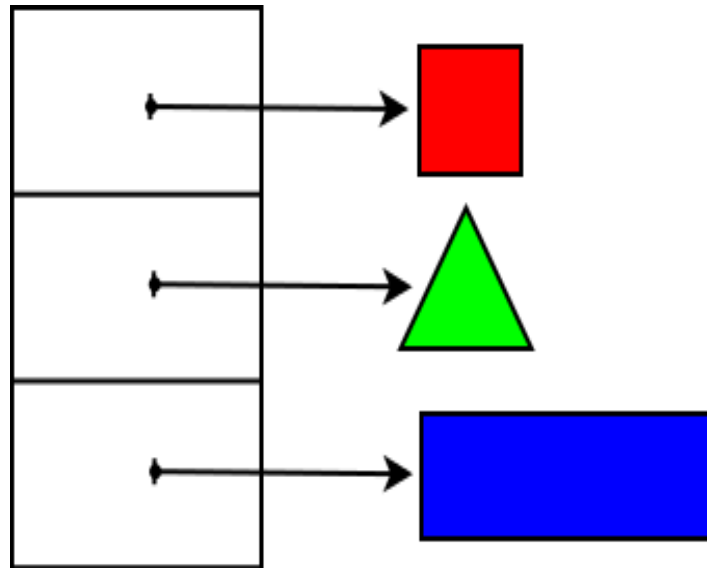
Polimorfismo

- Restricciones:



Polimorfismo

- Usos:
 - Estructuras de datos polimórficas
 - Pasaje de parámetros en funciones

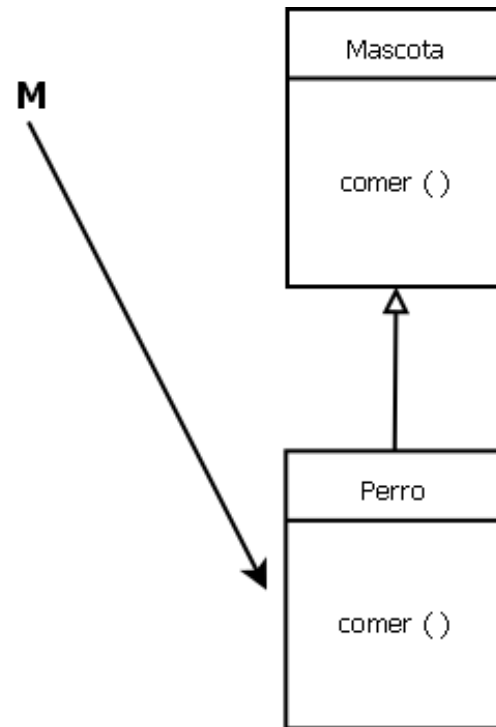


Polimorfismo

- Ligadura dinámica:

M : ref_mascota

- M := objeto_perro
- M.comer ()



Polimorfismo

```
17
18 // Metodos
19 public:
20     Rodado(string p, string m); // Constructor
21     string getPatente(); // Devuelve la patente
22     string getMarca(); // Devuelve la marca
23     virtual void mostrar(); // Muestra
24     virtual ~Rodado();
25 };
26 //
27 // Fin Clase Rodado
28 //
29
30 //
31 // Clase Auto
32 class Auto : public Rodado
33 {
34     // Atributos
35     private:
36         bool cajaAutomatica;
```



Métodos virtuales para que se produzca el polimorfismo con las clases herederas.

Polimorfismo

Java: por defecto ligadura dinámica

C++: por defecto ligadura estática

Para que se produzca el polimorfismo en C++ hay que indicar que el método es *virtual*.

Polimorfismo

- Funciones virtuales puras
 - Una *función virtual pura* es una función declarada en una clase base que no tiene definición relativa a la base. Como resultado, cualquier tipo derivado *debe definir su propia versión*.

```
virtual 'tipo' 'nombre_de_funcion'(lista_de_parametros) = 0
```

Polimorfismo

- Funciones virtuales puras
 - Ejemplo:

```
class figura
{
    double x, y;
public:
    void set_dim(double i, double j=0)
    {
        x = i;
        y = j;
    }

    virtual void mostrar_area() = 0; // pura
};
```

Polimorfismo

- Casteo dinámico
 - Sirve para que un puntero de una clase heredera pueda apuntar al mismo lugar que uno de una clase ancestral.

```
// Se muestran
for (int i = 0; i < 10; i++)
{
    rodados[i]->mostrar();

    Auto *pa = dynamic_cast<Auto*>(rodados[i]);
    if (pa)
        pa->mAuto();
    else
    {
        Camion *pc = dynamic_cast<Camion*>(rodados[i]);
        pc->mCamion();
    }
}
```

Casteo dinámico.

Si en la posición i del vector rodados hay un objeto de tipo Auto, realiza el casteo y devuelve la dirección que la toma pa.

De lo contrario devuelve NULL.

Fin
