Grado en Ingeniería Información

PROGRAMACIÓN II - Sesión 11

Tema 6.

Polimorfismo

Curso 2022-2023

Marta N. Gómez

UNIVERSIDAD NEBRIJA

UNIVERSIDAD NEBRIJA

T6. Polimorfismo

6.1. Definición

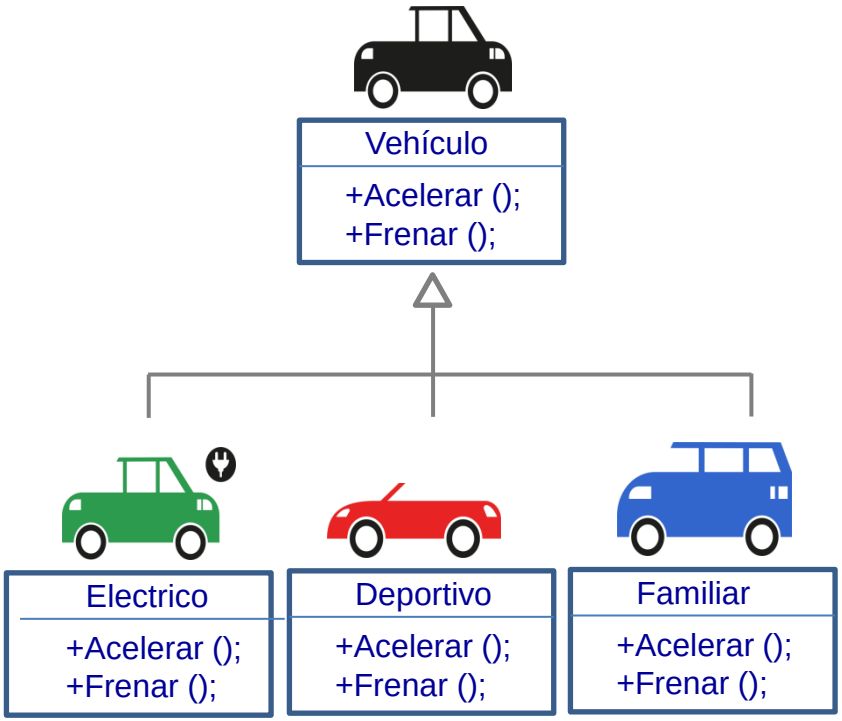
6.2. Tipos de Ligadura

6.3. Métodos Virtuales

6.4. Destructores Virtuales

6.5. Clases Abstractas

6.6. Conversión entre Objetos



ÍNDICE

UNIVERSIDAD NEBRIJA

T6. Polimorfismo

6.1. Definición

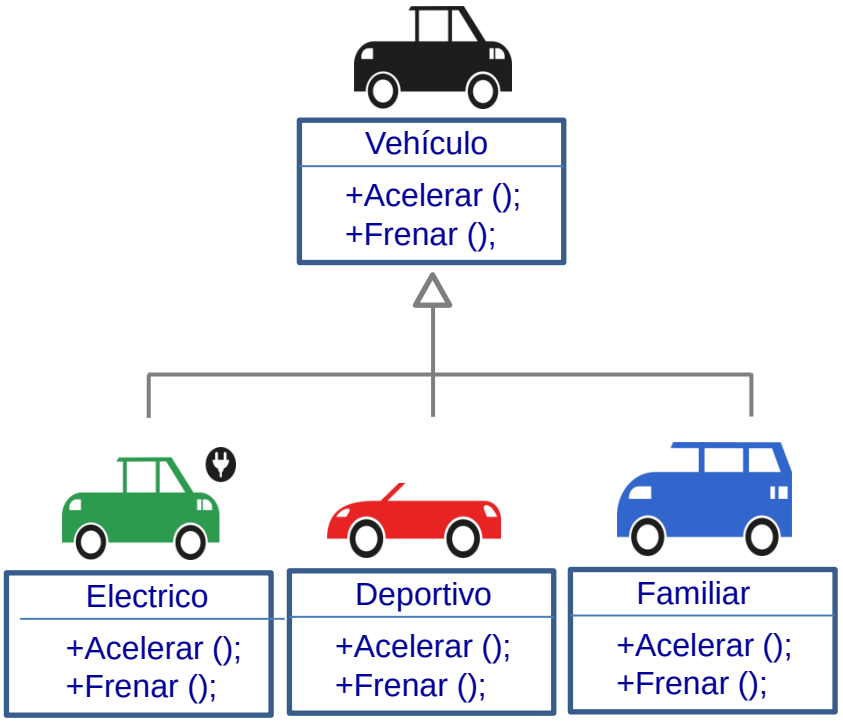
6.2. Tipos de Ligadura

6.3. Métodos Virtuales

6.4. Destructores Virtuales

6.5. Clases Abstractas

6.6. Conversión entre Objetos



ÍNDICE

Función virtual:

La definición de una función virtual en una clase base sirve como definición por defecto.

La función virtual se utilizará en toda clase derivada en la que la función no se redefina.

Clases Abstractas

```
class Persona {
    private:
        string nif;
        int edad;
        string nombre, apellidos;
    public:
        Persona():nif(""), edad(0), nombre(""), apellidos("") {}
        Persona(string const &identif, int aa, string const &nom,
                string const &apel):
                nif(identif), edad(aa), nombre(nom), apellidos(apel) {}
        Persona(Persona const &p):
                nif(p.nif), edad(p.edad), nombre(p.nombre),
                apellidos(p.apellidos) {}

        virtual ~Persona() {}

        virtual void mostrar() const;
        void okMatricula() const;
};
```

Clases Abstractas

```
class Estudiante : public Persona {
    private:
        int curso;
    public:
        Estudiante (string const &id, int a, string const &nom,
                    string const &ape, int cur):
                    Persona (id, a, nom, ape), curso(cur) {};

        void mostrar() const;
};

class Doctorando : public Estudiante {
    private:
        Persona direc;
    public:
        Doctorando (string const &id, int a, string const &nom,
                    string const &ape, int cur, Persona const &p):
                    Estudiante (id, a, nom, ape, cur), direc(p) {};

        void mostrar() const;
};
```

Clases Abstractas

```
//---------------CLASE PERSONA---------------------

void Persona::mostrar() const {
    cout << nombre << " " << apellidos << endl;
    cout << "\n\tNIF: " << nif << " EDAD: " << edad << endl;
}

void Persona::okMatricula() const {
     cout << "\n\n\tEl estudiante  ";
     mostrar();   // Ejecuta el método de la clase que lo llama
     cout << "\n\tHa sido matriculado en el curso correctamente.\n\n";
}

//---------------CLASE ESTUDIANTE---------------------

void Estudiante::mostrar() const {
    Persona::mostrar();

    cout << "\n\tEsta matriculado en el " << curso << " curso. ";
    cout << endl;
}

//---------------CLASE DOCTORANDO---------------------

void Doctorando::mostrar() const {
    Estudiante::mostrar();

    cout << "\n\tSu director es ";
    direc.mostrar();
}
```

Clases Abstractas

```
int main ()
{
    Estudiante estud ("123456789S", 20, "Eva","Sanz", 3);
    Persona person ("987654321P", 53, "Jose","Lozano");
    Doctorando doct ("12121212D", 30, "Ana","Vazquez", 2, person);

    cout << "\n\tDatos PERSONA\n";
    person.okMatricula();

    cout << "\n\tDatos ESTUDIANTE\n";
    estud.okMatricula ();

    cout << "\n\tDatos DOCTORANDO\n";
    doct.okMatricula();

    cout << "\n\n\t";
    return 0;
}
```

```
int main ()
{
    Estudiante estud ("123456789S"
    Persona person ("987654321P",
    Doctorando doct ("12121212D",

    cout << "\n\tDatos PERSONA\n";
    person.okMatricula();

    cout << "\n\tDatos ESTUDIANTE\
    estud.okMatricula ();

    cout << "\n\tDatos DOCTORANDO\
    doct.okMatricula();

    cout << "\n\n\t";
    return 0;
}
```

```
Datos PERSONA

El estudiante  Jose Lozano

NIF: 987654321P EDAD: 53

Ha sido matriculado en el curso correctamente.
```

```
Datos ESTUDIANTE

El estudiante  Eva Sanz

NIF: 123456789S EDAD: 20

Esta matriculado en el 3 curso.

Ha sido matriculado en el curso correctamente.
```

```
Datos DOCTORANDO

El estudiante  Ana Vazquez

NIF: 12121212D EDAD: 30

Esta matriculado en el 2 curso.

Su director es Jose Lozano

NIF: 987654321P EDAD: 53

Ha sido matriculado en el curso correctamente.
```

Referencias/punteros entre clase base y clases derivadas:

La clase base **sólo se puede acceder** a los atributos y métodos propios de la clase base.

Debido a la ligadura estática los **atributos** y **métodos** propios de los objetos de **clases derivadas son inaccesibles**.

El polimorfismo permite definir un puntero en una clase base y acceder a los miembros de su clase derivada u clases descendientes.

Así se logra definir una única interfaz para diferentes entidades/clases.

Clases Abstractas

```cpp
#include <iostream>
#include <cstring>
#include <vector>
#include <memory>

#include "Personas.h"

using namespace std;

int main ()
{
    vector<unique_ptr<Persona>> pUNI;

    Persona person ("987654321P", 53, "Jose","Lozano");

    pUNI.push_back(make_unique<Estudiante>("123456789S", 20, "Eva","Sanz", 3));
    pUNI.push_back(make_unique<Persona>(person));
    pUNI.push_back(make_unique<Doctorando>("12121212D", 30, "Ana","Vazquez", 2, person));

    for (unsigned int i{0}; i < pUNI.size(); i++){
        cout << "\n\tDatos PERSONA " << i+1 << "\n";
        pUNI.at(i)->okMatricula();
    }

    cout << "\n\n\t";
    return 0;
}
```

```cpp
#include <iostream>
#include <cstring>
#include <vector>
#include <memory>

#include "Personas.h"

using namespace std;

int main ()
{
    vector<unique_ptr<Persona>> pUNI;

    Persona person ("987654321P", 53, "Jose"

    pUNI.push_back(make_unique<Estudiante>("1
    pUNI.push_back(make_unique<Persona>(perso
    pUNI.push_back(make_unique<Doctorando>("1

    for (unsigned int i{0}; i < pUNI.size();
        cout << "\n\tDatos PERSONA " << i+1 <
        pUNI.at(i)->okMatricula();
    }

    cout << "\n\n\t";
    return 0;
}
```

```
Datos PERSONA 1

El estudiante  Eva Sanz

NIF: 123456789S EDAD: 20

Esta matriculado en el 3 curso.

Ha sido matriculado en el curso correctamente.

Datos PERSONA 2

El estudiante  Jose Lozano

NIF: 987654321P EDAD: 53

Ha sido matriculado en el curso correctamente.

Datos PERSONA 3

El estudiante  Ana Vazquez

NIF: 12121212D EDAD: 30

Esta matriculado en el 2 curso.

Su director es Jose Lozano

NIF: 987654321P EDAD: 53

Ha sido matriculado en el curso correctamente.
```

```
class Master : public Estudiante {
    private:
        string titulo;
    public:
        Master (string const &id, int a, string const &nom,
                string const &ape, int cur, string const &t):
                Estudiante (id, a, nom, ape, cur), titulo(t) {};
};


int main ()
{
    vector<unique_ptr<Persona>> pUNI;

    Persona person ("987654321P", 53, "Jose","Lozano");

    pUNI.push_back(make_unique<Estudiante>("123456789S", 20, "Eva","Sanz", 3));
    pUNI.push_back(make_unique<Persona>(person));
    pUNI.push_back(make_unique<Doctorando>
                ("12121212D", 30, "Ana","Vazquez", 2, person));
    pUNI.push_back(make_unique<Master>
                ("34343434M", 28, "Carlos","Jimenez", 1, "Bioinformatica"));

    for (unsigned int i{0}; i < pUNI.size(); i++){
        cout << "\n\tDatos PERSONA " << i+1 << "\n";
        pUNI.at(i)->okMatricula();
    }
```

```
class Master : public Estudiante {
    private:
        string titulo;
    public:
        Master (string const &id, int a, st
                string const &ape, int
                Estudiante (id, a, nom
};


int main ()
{
    vector<unique_ptr<Persona>> pUNI;

    Persona person ("987654321P", 53, "Jose","Loz

    pUNI.push_back(make_unique<Estudiante>("12345
    pUNI.push_back(make_unique<Persona>(person));
    pUNI.push_back(make_unique<Doctorando>
                   ("12121212D", 30, "Ana","Vazqu
    pUNI.push_back(make_unique<Master>
                   ("34343434M", 28, "Carlos","Ji

    for (unsigned int i{0}; i < pUNI.size(); i++)
        cout << "\n\tDatos PERSONA " << i+1 << "\
        pUNI.at(i)->okMatricula();
    }
```

```
Datos PERSONA 1

El estudiante  Eva Sanz
NIF: 123456789S EDAD: 20
Esta matriculado en el 3 curso.
Ha sido matriculado en el curso correctamente.

Datos PERSONA 2

El estudiante  Jose Lozano
NIF: 987654321P EDAD: 53
Ha sido matriculado en el curso correctamente.

Datos PERSONA 3

El estudiante  Ana Vazquez
NIF: 12121212D EDAD: 30
Esta matriculado en el 2 curso.
Su director es Jose Lozano
NIF: 987654321P EDAD: 53
Ha sido matriculado en el curso correctamente.

Datos PERSONA 4

El estudiante  Carlos Jimenez
NIF: 34343434M EDAD: 28
Esta matriculado en el 1 curso.
Ha sido matriculado en el curso correctamente.
```

CLASE ABSTRACTA

- Se trata de clases que **representan conceptos abstractos** para los que **no es necesario crear objetos**.
- **No** tienen una **función concreta** dentro de la aplicación.
- No se pueden **crear objetos de una clase abstracta**.
- Se utiliza para agrupar una **serie de atributos comunes** a otras clases que derivan de ella a través de la herencia.
- Sirve como **clase base** para otras clases.

CLASE ABSTRACTA

- Se definen a través de uno o varios **Métodos Virtuales Puros**
- **Método Virtual Puro es un Método Virtual** que **no se implementa en la clase base** o superclase, **sólo se declara el prototipo seguido de = 0**.

Sintaxis del método virtual puro:

virtual tipoValorRetorno* nombreMetodo *(parámetros) = 0;

CLASE ABSTRACTA

- Los **métodos virtuales puros obligan** a que las **clases derivadas tengan que implementar el código para cada uno de los métodos virtuales puros**.
- **Si las clases derivadas no incluyen la implementación** de los métodos virtuales puros, también son **clases abstractas**.
- Los **métodos virtuales puros** son útiles porque hacen **explícita** la **abstracción de una clase**.
- Una **clase abstracta** puede tener **métodos virtuales puros** y **métodos normales**.

```
class Base
{  private:
        ...
   public:
      virtual void imprimir () = 0;
};

class Derivada: public Base  {
   private:
        ...
   public:
      void imprimir ()
          { cout << "Clase Derivada" << endl; }
};
```

Clases Abstractas

```
class Padre {
    public:
        Padre(){}
        virtual ~Padre();
        virtual void mostrar() const = 0;
        virtual void escribe() const { cout << "\n\tSoy una clase ";}
};

class Hija: public Padre {
    string nom;
    public:
        Hija(){}
        void mostrar() const {cout << "\n\tMOSTRAR DE HIJA\n\n";}
        void escribe() const {
            Padre::escribe();
            cout << "HIJA\n\n";  }
};

int main() {
    Hija h1;

    h1.escribe();
    h1.mostrar();

    return 0;
}
```

Clases Abstractas

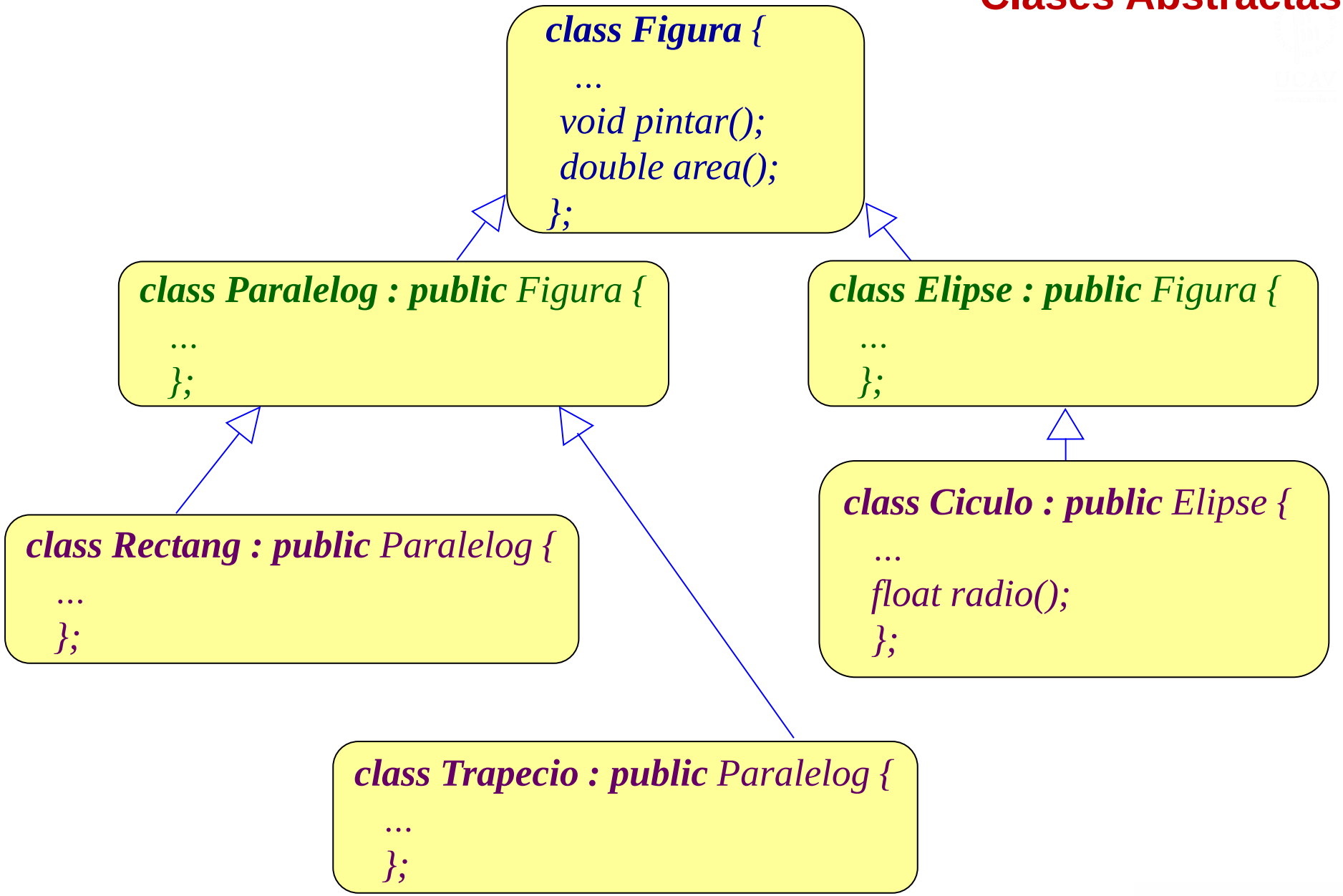
```
class Padre {
    public:
        Padre(){}
        virtual ~Padre();
        virtual void mostrar() const = 0;
        virtual void escribe() const { cout << "\n\tSoy una clase ";}
};

class Hija: public Padre {
    string nom;
    public:
        Hija(){}
        void mostrar() const {cout << "\n\tMOSTRAR DE HIJA\n\n";}
        void escribe() const {
            Padre::escribe();
            cout << "HIJA\n\n";  }
};

int main() {
    Hija h1;

    h1.escribe();
    h1.mostrar();

    return 0;
}
```

```
Soy una clase HIJA


MOSTRAR DE HIJA
```

Clases Abstractas

```
class Padre {
    public:
        Padre(){}
        virtual ~Padre();
        virtual void mostrar() const = 0;
        virtual void escribe() const { cout << "\n\tSoy una clase ";}
};

class Hija: public Padre {
    string nom;
    public:
        Hija(){}
        void mostrar() const {cout << "\n\tMOSTRAR DE HIJA\n\n";}
        void escribe() const {
            Padre::escribe();
            cout << "HIJA\n\n";  }
};

int main() {
    Hija h1;
    Padre p1;     Variable type 'Padre' is an abstract class

    h1.escribe();
    h1.mostrar();

    return 0;
}
```

Clases Abstractas



Clases Abstractas

```
class Figura    {
    public:
        Figura(){};
        virtual ~Figura(){};
        virtual void pintar() const = 0;
};

class Elipse: public Figura     {
    private:

    public:
        void pintar () const { cout << "\n\tPintar Elipse\n\n"; }
};
```

Clases Abstractas

```
int main()
{
    shared_ptr<Figura> pfigura=make_unique<Elipse>();
    Elipse elipse;

    elipse.pintar();

    pfigura->pintar();
    return 0;
}
```

```
        Pintar Elipse


        Pintar Elipse

Press <RETURN> to close this window...
```

UNIVERSIDAD NEBRIJA

T6. Polimorfismo

6.1. Definición

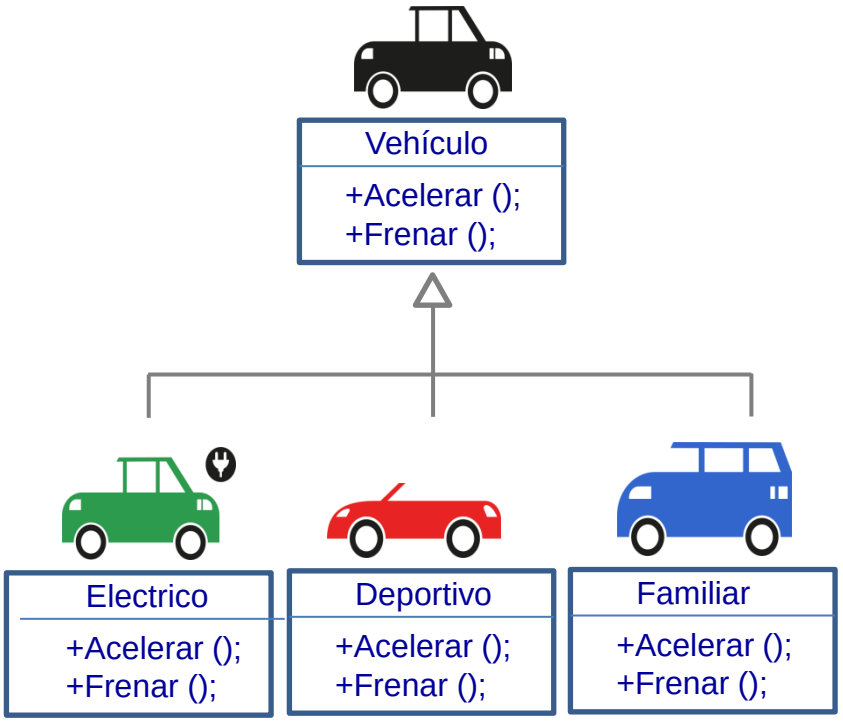
6.2. Tipos de Ligadura

6.3. Métodos Virtuales

6.4. Destructores Virtuales

6.5. Clases Abstractas

6.6. Conversión entre Objetos



ÍNDICE

Las funciones derivadas, normalmente, añaden más funcionalidad a la clase base, pero **la clase base no puede acceder a estos métodos**.

Downcasting:

- Convertir un tipo de la **clase padre** en un tipo de la **clase derivada para poder acceder así a sus miembros**.

Conversión entre objetos - Downcasting

Condiciones para poder hacer **downcasting:**

- La **clase derivada hereda de la clase base/padre.**
- La **clase derivada** es del **tipo que se indica en tiempo de ejecución**.

Sintaxis:

claseDerivada* identif = dynamic_cast<claseDerivada*>

*(ptr_claseBase.**get()**);*

Donde aparece:

- Una variable de tipo **puntero clásico**: claseDerivada* ***identif***
- Casting o conversión de tipos: ***dynamic_cast***
- Se indica el tipo a obtener: ***<claseDerivada*>***
- Se obtiene el puntero clásico asociado al **puntero inteligente *ptr_claseBase*** a través de: *ptr_claseBase.**get()***

Finalmente, ***identif*** es del tipo ***claseDerivada*** y se puede acceder a sus miembros.

Conversión entre objetos - Downcasting

Si se intenta hacer una conversión entre objetos que **no sea posible**, **se produce error en tiempo de ejecución**, aunque el programa **compila correctamente.** Esto se debe a que ***dynamic_cast*** se hace de forma **dinámica,** es decir, **en durante la ejecución**.

El motivo es que **la conversión no se ha realizado** y el puntero valdrá ***nullptr***, luego no se podrá acceder al miembro de la clase.

Solución:

- Hay que comprobar que el valor del puntero no sea ***nullptr***:

  if *(identif != nullptr) …*

Conversión entre objetos - Downcasting

```cpp
#include <iostream>
#include <memory>
#include <vector>

using namespace std;

class Figura   {
    public:
        Figura(){};
        virtual ~Figura(){};
        virtual string getTipo() const = 0;
        virtual float calcularArea() const = 0;
};

class Cuadrado: public Figura    {
    private:
        float lado;
    public:
        Cuadrado(float a):lado(a){}
        string getTipo () const { return "Cuadrado"; }

        float calcularArea() const { return lado*lado; }
        float getLado() const { return lado; }
};
```

Conversión entre objetos - Downcasting

```
class Rectangulo: public Figura    {
    private:
        float ladoG, ladoP;
    public:
        Rectangulo(float a, float b):ladoG(a), ladoP(b){}
        string getTipo () const { return "Rectangulo"; }

        float calcularArea() const { return ladoG*ladoP; }
        float getLadoG() const { return ladoG; }
        float getLadoP() const { return ladoP; }
};
```

Conversión entre objetos - Downcasting

```
int main(){
  vector<unique_ptr<Figura>> misFiguras;

  misFiguras.push_back(make_unique<Rectangulo>(4,6));
  misFiguras.push_back(make_unique<Cuadrado>(5));
  misFiguras.push_back(make_unique<Cuadrado>(2));

  for(auto const & elem: misFiguras){
    if(elem->getTipo() == "Rectangulo"){
       Rectangulo* t = dynamic_cast<Rectangulo*>(elem.get());
       if (t) {
          cout << "\n\tEl lado grande mide: " << t->getLadoG() << " cm\n\n"
       }
    }
  }

  return 0;
}
```

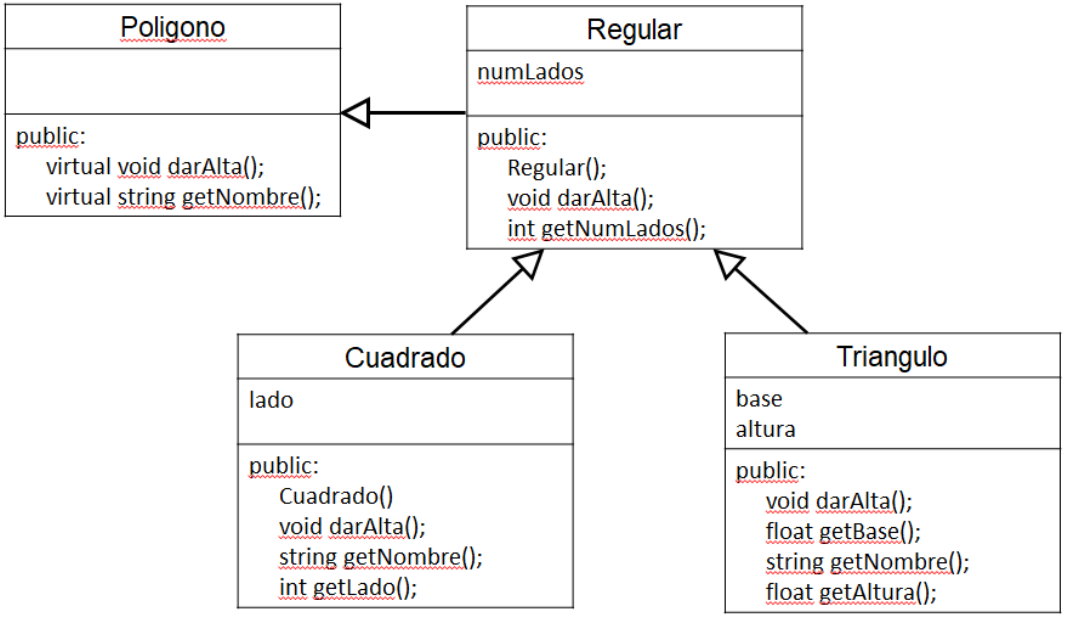
t es un puntero de la clase ***Retangulo*** obtenido al convertir el puntero inteligente ***elem*** de la clase ***misFiguras*** para poder acceder a los miembros de ***Rectangulo***.

Conversión entre objetos - Downcasting

```
int main(){
  vector<unique_ptr<Figura>> misFiguras;

  misFiguras.push_back(make_unique<Rectangulo>(4,6));
  misFiguras.push_back(make_unique<Cuadrado>(5));
  misFiguras.push_back(make_unique<Cuadrado>(2));

  for(auto const & elem: misFiguras){
    if(elem->getTipo() == "Rectangulo"){
       Rectangulo* t = dynamic_cast<Rectangulo*>(elem.get());
       if (t) {
          cout << "\n\tEl lado grande mide: " << t->getLadoG() << " cm\n\n";
       }
    }
  }

  return 0;
}
```

```
        El lado grande mide: 4 cm

Press <RETURN> to close this window...
```

EJERCICIO



Realizar un programa que responda al diagrama anterior y que presente un menú con las siguientes opciones: **(1)** Dar de alta cuadrado; **(2)** Dar de alta triangulo; **(3)** Mostrar poligonos regulares y **(0)** SALIR.

Los polígonos se deberán almacenar en un vector. Los métodos son:

Método ***darAlta***, debe permitir al usuario introducir los atributos de cada clase.

Método ***getNombre***, según corresponda, devolverá un ***string*** con el nombre de cada polígono regular (cuadrado, triángulo, etc.)

Los métodos ***gets*** restantes permitirán obtener el valor de los atributos correspondientes a cada clase.