Programación II

Tema 6

jtorresmat@nebrija.es

Índice

Recapitulando antes de la recta final...

- ¿Qué hemos visto hasta el momento?
  - Clases (constr., destr., métodos, etc)
  - Excepciones
  - Sobrecarga
    - Funciones/métodos
    - Operadores
  - Plantillas
    - Funciones
    - Clases

Recapitulando antes de la recta final...

- Características de la POO:
  - Abstracción: es el mecanismo de diseño en POO. Nos permite extraer datos y comportamientos comunes de entidades para representarlos en clases
    - Ejemplo: mesa, vehículo, etc.

Recapitulando antes de la recta final...

- Características de la POO:
  - Encapsulamiento: es una técnica que nos permite que cada clase se comporte como una caja negra que nos permita manejar sus objetos como unidades básicas. Desde el exterior serán elementos con una interfaz pública, que le permitirá comunicarse con su entorno.

Recapitulando antes de la recta final...

- Características de la POO:
  - Herencia: es un mecanismo que nos permite crear clases derivadas(más concretas) a partir de otras clases (más generales)
    - Ejemplo:
      - Clase base vehículo: pasajeros, carga, etc
      - Clase derivada avión: número de motores

Recapitulando antes de la recta final...

- Características de la POO:
  - Polimorfismo: esta característica nos permitirá disponer de múltiples implementaciones de un mismo método dependiendo de la clase donde se realice. Tendremos acceso a distintos métodos con un mismo nombre. Pero a esto… ya volveremos más adelante :)

Recapitulando antes de la recta final...

- ¿Qué nos queda por ver?
  - Herencia y polimorfismo

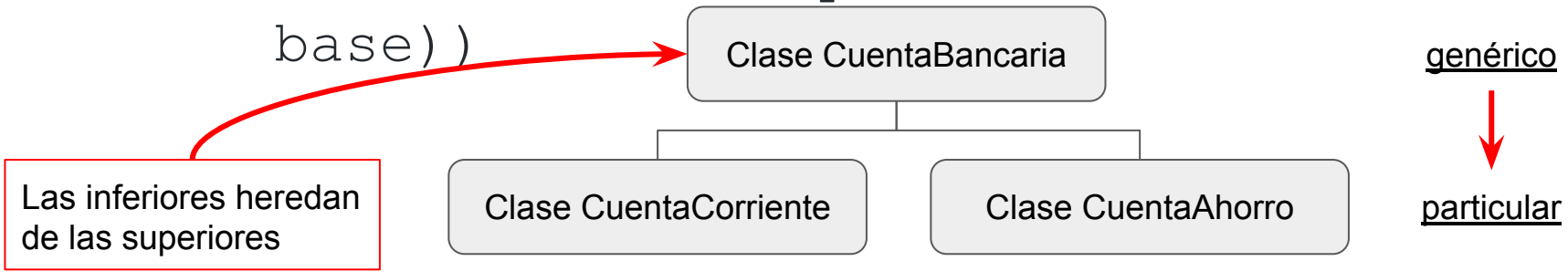
Índice

Herencia

- Herencia:
  - Es una de las características más importantes de la POO, junto a:
    - Abstracción
    - Encapsulación y ocultación
    - Polimorfismo
  - Permite que una clase herede los métodos y atributos de otra clase (los miembros de la clase “padre” pertenecen a la clase que hereda de ella)

Herencia

- Herencia:
  - Con la herencia se consigue que todas las clases estén clasificadas jerárquicamente(Clase base->Clases derivadas (las que heredan de la base))



Herencia

- Heredar implica que las subclases disponen de “casi-todos” los métodos y variables de su superclase. Pero podemos añadir más miembros => Extienden la funcionalidad de la clase base.
    - Herencia simple: una clase tiene una superclase
    - Herencia múltiple: una clase tiene más de una superclase (cuidado con ambigüedades)

Herencia

- Ejemplo herencia clase cuenta:

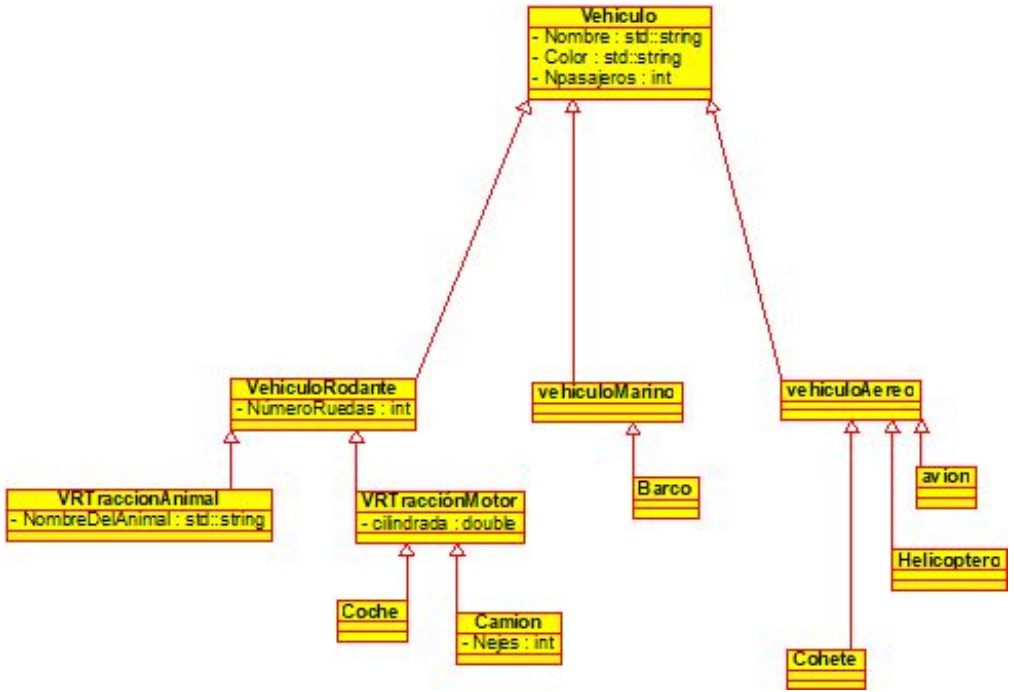
  código

Herencia

- Ejemplo herencia:
  - Plantear un diagrama de clases en el que a partir de una clase vehículo vayamos particularizando haciendo uso de la herencia.

Herencia

Vehículo



Herencia

- La herencia es un mecanismo para la reutilización y mantenibilidad del código
- Consiste en ir construyendo tipos de datos (clases) cada vez más concretos. La filosofía de diseño usando mecanismo de herencia es pasar de lo general a lo particular (ejemplo anterior, vamos de vehículo como concepto general a vehículo rodante, coche,...)

Herencia

- Ejemplos de arranque:
  - Implementar una clase suma y otra multiplicación para 2 números, que hereden de una clase Operación. código
    - ¿Que tienen en común + y *?
      - Operaciones sobre 2 números
        - Cargar 2 valores
        - Hacen una operación
      - Dan un resultado

Herencia

- Ejemplos de arranque:
  - Queremos hacer un tipo de datos que almacene info de:
    - Triángulo
    - Cuadrado
    - ...

Herencia

- ¿Qué tienen en común?=>Son figuras geométricas regulares
- Al ser las tres figuras del mismo tipo... ¿Qué características tienen en común?
    - Se componen de lados => n° de lados
    - El lado es un segmento => Tiene longitud
- ¿Qué funciones pueden compartir?:
    - Cálculo del perímetro (n° de lados * Longitud)
    - Cálculo del área (perímetro*apotema/2)
- ¿Tienen características diferentes?
    - Triángulo tiene altura.
    - Cuadrado tiene diagonal.
    - etc...

Apotema: distancia mínima del centro a uno de los lados

Herencia

- Primera aproximación para clase figura geométrica regular

```
class FigGeomRegular {
      public:
            FigGeomRegular(float _long, float _lados);
            float getArea() const;
            float getPerimeter() const;

      private:
             int n_lados;
             float longitud;
 };
```

Herencia

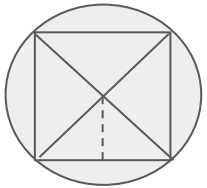
- Primera aproximación para Clase triángulo equilátero

```
class TrianguloEq: public FigGeomRegular{
    public:
      TrianguloEq(float _long);
      float getHeight() const;
 };
```

- Completad para hacer el ejercicio 8 de la colección de problemas de github:
  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/EJERCICIOS.md

Herencia

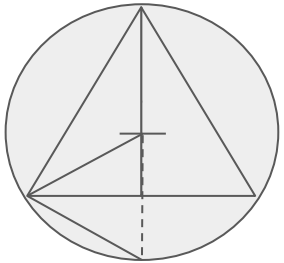
- Ejemplo herencia:
  - Ejercicio 8 colección: código



Lado=$\sqrt{(radio^2+radio^2)}$
r=L/$\sqrt{2}$

apotema=$\sqrt{[r^2+(L/2)^2]}$

Despejando… :
ap=L/2



$r^2=((L/2)^2+(r/2)^2)$
r=L/$\sqrt{3}$

$ap^2=r^2-(L/2)^2$

Despejando… :
ap=L/$\sqrt{12}$

https://www.superprof.es/apuntes/escolar/matematicas/geometria/basica/apotema-de-la-piramide.html#tema_apotema-del-triangulo-equilatero
https://www.superprof.es/apuntes/escolar/matematicas/geometria/basica/aplicaciones-del-teorema-de-pitagoras-iv-lado-de-un-triangulo-equilatero-y-de-un-cuadrado.html

Herencia

- "Reglas" de la herencia:
  - Una subclase hereda todos los métodos y variables de su superclase, excepto constructores. Veremos todas las reglas de acceso a continuación… pero a modo de "entrante":
    - La subclase no tiene acceso a variables miembro privados de su superclase
    - La subclase si tiene acceso a variables miembro públicos de su superclase
  - Una subclase puede añadir sus propios miembros(atributos y métodos). Si algún nombre coincide con el de la superclase, este queda oculto para la subclase.
  - Los miembros heredados por una subclase pueden ser heredados a su vez por subclases de ella misma (Propagación de herencia) [Primer ejemplo para ver a que tenemos acceso]

Herencia

- Herencia:
  - Sintaxis para derivar un clase

```
class <clase_derivada> : [public | private | protected] <clase_base> {};
```

  - Control de acceso a la clase base:
    - public
    - private
    - protected

Herencia

- ¡¡¡**Cuidado**!!! tenemos 2 sitios donde usamos public, private y protected…
    - En la declaración de la clase (como vimos en el tema de clases)
    - En el control de acceso a la clase base (como estamos viendo en el tema de herencia)

Herencia

- Modificadores de acceso (en declaración de clase):
  - Si no se indica ninguno, por defecto es private.
  - Private: los elementos declarados así son accesible únicamente dentro de la propia clase (no pueden ser accedidos por métodos de otras clases ni por funciones externas, como por ejemplo desde el main())
  - Public: los elementos declarados así son accesibles desde dentro y fuera de la misma clase (cualquier parte del programa)
  - Protected: son igual que los *private* para funciones externas o métodos de otras clases, pero actúa como *public* para los métodos de sus subclases.

Herencia

- Modificadores de acceso (en declaración de clase):

Tipo acceso	Desde la propia clase	Desde el exterior
public:	Si	Si
private:	Si	No
protected:	Si	No

```
class MiClase{
public:
      int miVariablePublica;
private:
      int miVariablePrivada;
protected:
      int miVariableProtegida;};
```

A "protected" no le hemos prestado atención hasta ahora, ya que sin usar el mecanismo de herencia su comportamiento es como el de los miembros "private"

Herencia

- Control de acceso en herencia(acceso a clase base):

Acceso	Descripción	
public	Miembros públicos de clase base son públicos para la clase derivada Miembros privados de clase base son privados para la clase derivada Miembros protegidos de clase base son protegidos en la clase derivada	Mantenemos igual
private	Miembros públicos de clase base son privados para la clase derivada Miembros privados de clase base no son accesibles para la clase derivada Miembros protegidos de la clase base son privados para la clase derivada	Público y protegido a privado
protected	Miembros públicos de clase base son protegidos de la clase derivada Miembros privados de clase base no son accesibles para la clase derivada Miembros protegidos de la clase base son protegidos de la clase derivada	Público a protegido

```
class <clase_derivada> : [public | private | protected] <clase_base> {};
```

Herencia

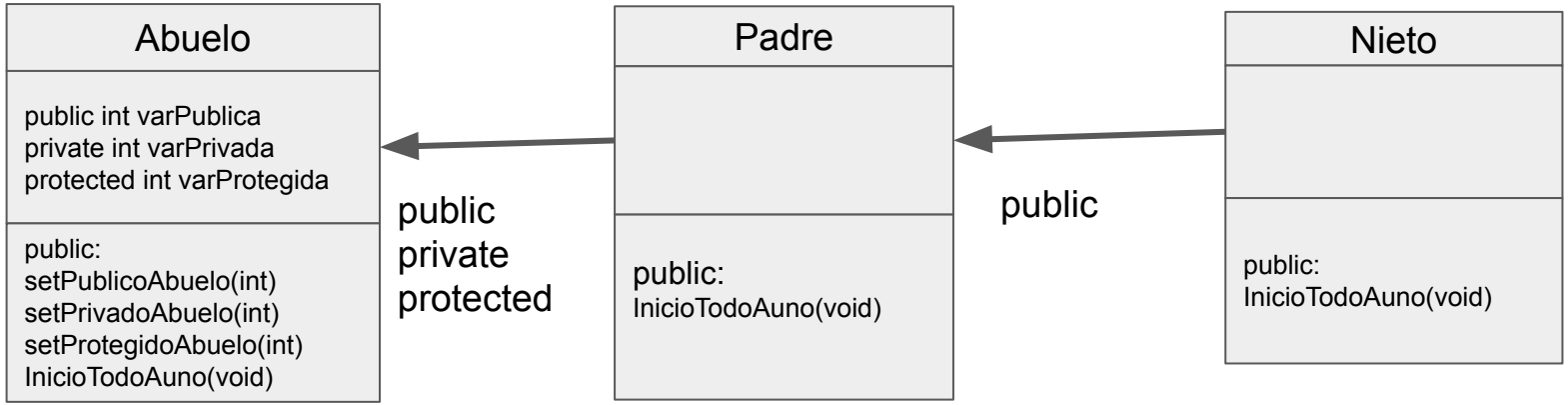
- Control de acceso en herencia(acceso a clase base):
  - La tabla anterior era un resumen…
  - Ahora algo más en detalle

Herencia

- Ejemplo: control acceso en herencia

Herencia

- Ejemplo: abuelo-padre-hijo



Implementar Nieto::InicioTodoAuno(void) para las herencias entre padre y abuelo public, protected y private

Herencia

- Herencia (“más cosas”):
  - Dentro de una clase derivada podemos añadir más datos y métodos. Y dentro de esos nuevos métodos podremos llamar a los métodos y datos de la clase base, según los criterios del control de acceso que tengan.
  - ¿Se hereda todo? => **NO**
    - No se heredan constructores
    - No se heredan destructores
    - No se heredan funciones/clases amigas
    - No se hereda operador de asignación sobrecargado

Herencia

- Herencia - constructores/Destructores:
  - Constructores y destructores no son heredados por las clases derivadas. Pero los miembros de la clase base deben ser inicializados => El constructor de la clase base debe ser llamado por el de la clase derivada
  - Sintaxis para constructor de clase derivada (ejemplo):

```
class TrianguloEq : public FigGeomRegular{
public:
  TrianguloEq(float _long):FigGeomRegular(_long, 3){
        altura=sqrt(pow(_long,2)-pow(_long/2,2));
  }
...
}
```

Herencia

- Herencia - constructores/Destructores:
  - En el caso de los constructores por defecto (constructores sin parámetros) no es necesario hacer la llamada de manera explícita desde la clase derivada. Al instanciar un objeto de la clase derivada haciendo uso de dicho constructor primero se ejecutará el constructor por defecto de la clase base y luego el de la derivada.

    Ejemplo

Herencia

- Herencia - constructores/Destructores:
  - Al declarar un objeto de una clase derivada el compilador ejecuta en primer lugar el constructor de la clase base y después el de la derivada.
  - Jerarquía constructores: Si la clase derivada tuviera objetos miembros, sus constructores se ejecutarían después del constructor de la clase base y antes del constructor de la clase derivada
  - Jerarquía destructores: se llaman en orden inverso a la derivación, es decir de clase derivada hacia clase base (el último destructor llamado es el de la clase base)

Herencia

- Herencia - constructores/Destructores:
  - Ejemplo orden constructores (¿Qué muestra?):

```
#include <iostream>
class Padre{
  public:
    Padre(int a){std::cout << "Constructor Padre"<<a<<std::endl;}
};
class Hija: public Padre{
  public:
    Hija():Padre(1){std::cout << "Constructor Hija"<<std::endl;}
};
class Nieta: public Hija{
  public:
    Nieta(){std::cout<< "Constructor Nieta"<<std::endl;}
};
```

```
int main()
{
  Nieta miClase;
}
```

Herencia

- Herencia - constructores/Destructores:
  - Ejemplo orden destructores (¿Qué muestra?):

```
#include <iostream>
#include <memory>

class Padre{
 public:
      Padre(){std::cout << "Constructor Padre"<<std::endl;}
      ~Padre(){std::cout << "Destructor Padre"<<std::endl;}};

class Hija: public Padre{
   public:
      Hija(){std::cout << "Constructor Hija"<<std::endl;}
      ~Hija(){std::cout << "Destructor Hija"<<std::endl;}};

class Nieta: public Hija{
   public:
      Nieta(){std::cout<< "Constructor Nieta"<<std::endl;}
      ~Nieta(){std::cout<< "Destructor Nieta"<<std::endl;}};
```

```
int main()
{
  std::shared_ptr<Nieta> miPunt;
  std::cout << "Aun no apunto a ningún sitio...\n";
  miPunt = std::make_shared<Nieta>(); // Inicializo el puntero
  std::cout << "hago cosas\n";

  std::cout << "Terminando...\n";
  return 1;
}
```

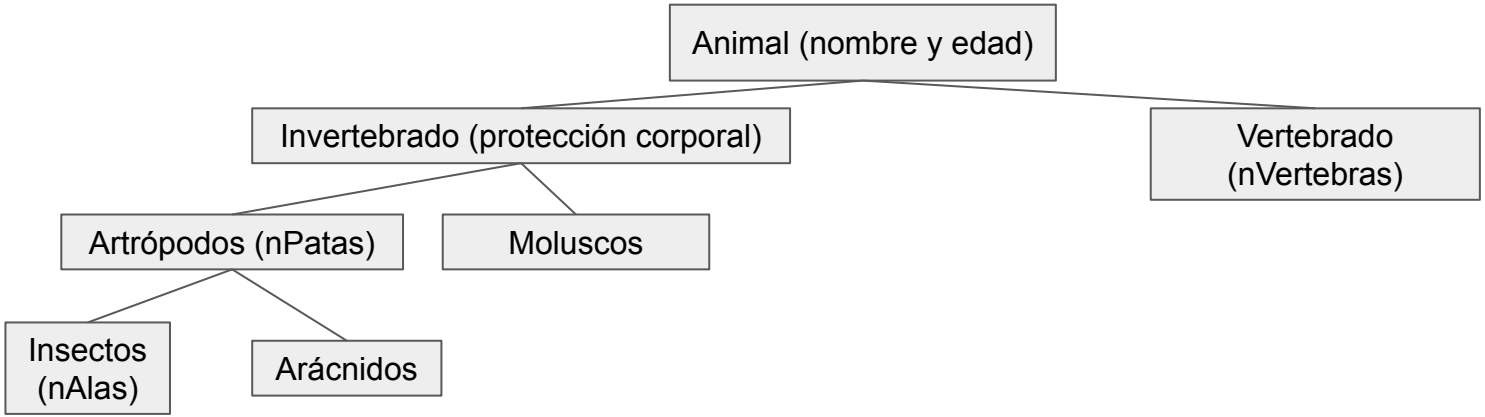
Herencia: Github

- Concepto de herencia: Herencia
- Constructores y destructores: Constr./Destruct. Herencia
- Control de acceso: Herencia. Miembros públicos, protegidos y privados

Índice

Ejemplos

Implementar las clases necesarias haciendo uso de herencia para representar la siguiente clasificación



Código