Tema 4. POO. Aspectos Avanzados

a) Herencia Múltiple
b) Control de copia y gestión de recursos: clonar, mover, comparar
c) El constructor de copia y el operador de asignación vía copia
d) Variable dinámica new

Herencia

- La herencia simple consiste en la herencia de una sola superclase.
- En ejercicios anteriores, hemos hecho que una clase B heredase de otra clase superior A.
- Pero también es posible crear una clase C que herede solamente de B, así esta clase C hereda todos los comportamientos de B y, a su vez, todos los de A.
- Este tipo de herencia sigue siendo simple, aunque se denomina herencia en cadena.
- Se denomina al conjunto de clases con sus herencias como **jerarquía de herencias**.

Herencia en cascada

- Se recomienda utilizar herencias en cascada cuando la clase intermedia (Perro) no posee todas las cualidades de un objeto que también “es-un” objeto de esta clase (PerroPolicia), por lo que hará falta definir el comportamiento especifico en una sub-sub-clase.

AnimalTerrestre
nombre: string # especie: string
+ *AnimalTerrestre* () + *AnimalTerrestre* (nombre: string, especie: string) + *comer(comida: string): string = 0* + *toString(): string* + *andar(): string*

Perro
duenho: string # raza: string
+ Perro() + Perro (…) + *comer(comida: string): string* + *toString(): string* + *ladrar(): string*

PerroPolicia
- anhos_servicio: int
+ PerroPolicia () + PerroPolicia (..) + *toString(): string* + *ladrar(): string* + *buscarObjeto(objeto:string): bool*

Herencia en cascada

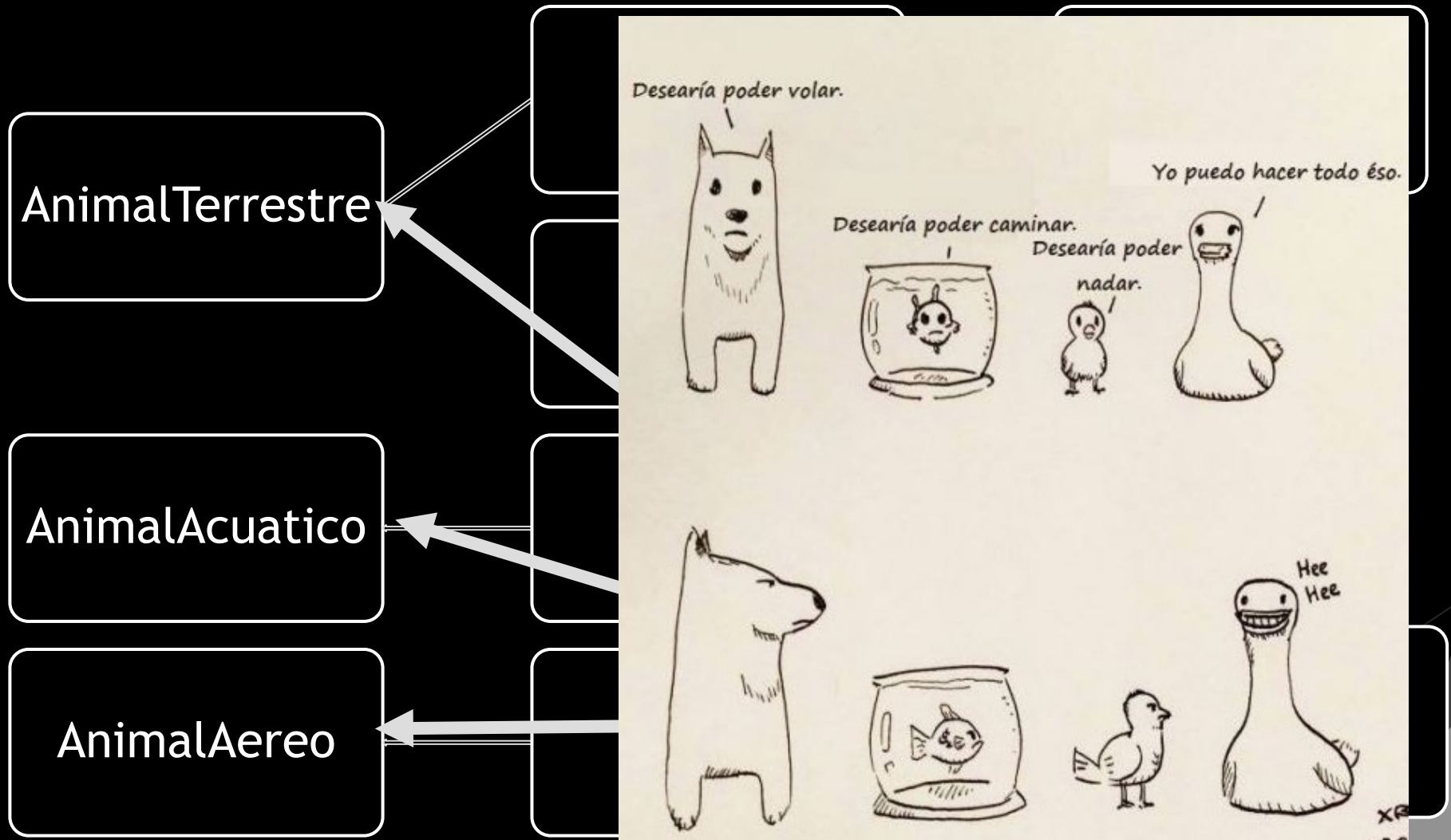
- La herencia es en cadena, por lo que todos los atributos y métodos de la superclase se encuentran en la subclase y en la clase final (sub-sub-clase)
- Los constructores, por supuesto, también se encadenan, por lo que un PerroPolicia debe construir un Perro, no un AnimalTerrestre
- La virtualización de métodos nos ha permitido que seleccionemos cuales funciones pueden sobreescribirse en sub-clases.
- Las demás funciones se utilizan de forma normal.

Herencia en cascada

- Herencia en cascada se define solamente para sub-clases que son, a su vez, sub-clases de otra superior.
- Claramente, las clases finales pueden ser muy dispares, pero la similitud entre clases (los comportamientos y atributos) aumentan a medida que hablamos de clases más generales, es decir que estén más arriba en la jerarquía.
- En este ejemplo, las 3 clases “superiores” en realidad podrían describirse de acuerdo con una aún más general, la clase Animal

Herencia múltiple

- Pero hay situaciones en que existen clases que deben heredar de múltiples Clases Padre: esto se denomina herencia múltiple.



Herencia múltiple

- Para crear Clases derivadas con dos o más clases Base, hay que especificar durante la definición separando por comas cada clase Base, en conjunto con los niveles de acceso garantizados.

#pragma once

```
class Pato: public AnimalTerrestre, public AnimalAcuatico, public AnimalAereo {
```

- Como los atributos y métodos son completamente heredados, ha de tener cuidado a la hora de heredar de 2 clases porque lleva a problemas de ambigüedad.
- La solución siempre es sobreescribir, usando virtual, **todos** los métodos conflictivos como por ejemplo el método toString.
- En caso de variables ambiguas, también se puede volver a declarar variables con el mismo nombre en la clase de implementación, pero la solución siempre consiste en resolver ámbitos para leer/escribir atributos en clases específicas.

```
(con resolucion de ambito):\nNombre: " + AnimalAcuatico::nombre + ". Especie: " + AnimalTerrestre::especie +
```

Herencia múltiple

- En el ejemplo, pato es-un animal acuático, pero también es-un animal terrestre, por estas razones pato hereda de ambas clases y los métodos de ambas funciones y clases se encuentran disponibles para su uso en cualquier objeto de esta clase.

```
Pato pato = Pato("Pato", "Anas Platyrhynchos", "Corredor", "Verde");

cout << pato.comer("maiz") << endl;
cout << pato.hacerCosasDePato() << endl;
cout << pato.toString() << endl;

cout << pato.andar() << endl;
cout << pato.nadar() << endl;
```

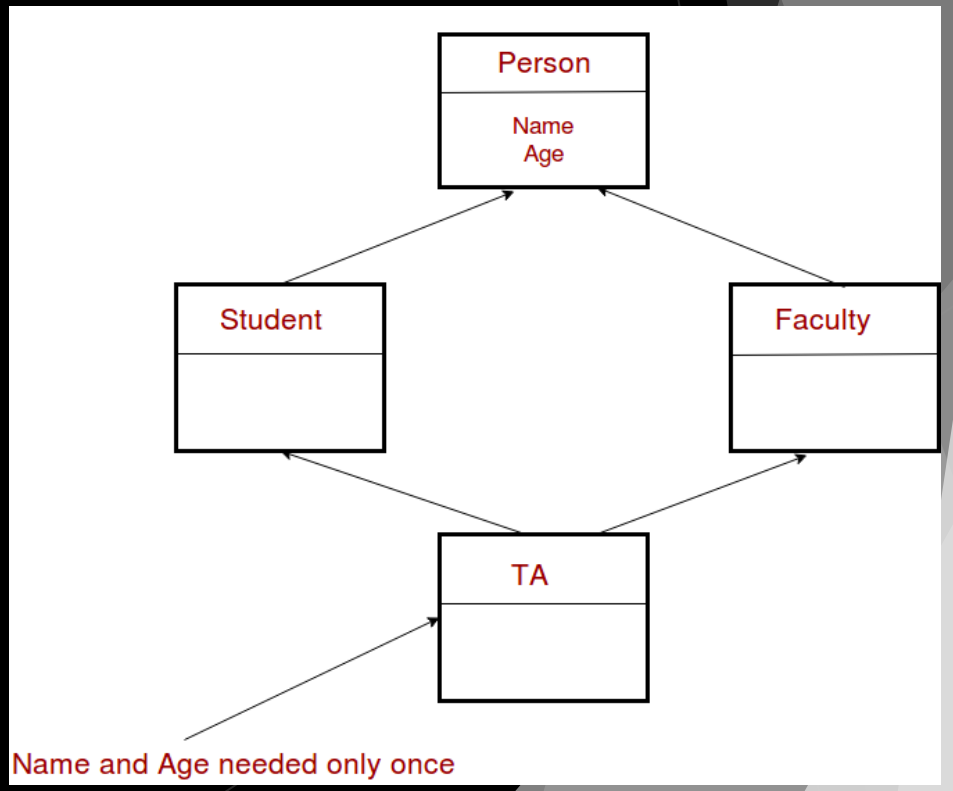
Herencia múltiple

- Tanto como para un Pato como para muchas otras clases, existen situaciones en que las clases heredadas son, a su vez, especificaciones de una clase más general.
- Esto genera el “problema del diamante” que es una extensión del problema de la función toString.
- Por lo que la solución se encuentra en virtualizar las funciones para prevenir posibles errores.
- Pero más importante es que los atributos no se deben inicializar más que una vez.
- Para ello extendemos el uso de la palabra reservada virtual para las clases y definir que las herencias también pueden ser virtuales.

class Student: virtual public Person {

class Faculty: virtual public Person {

class TA: public Student, public Faculty {



Herencia múltiple

- La solución más eficiente siempre será virtualizar las clases intermedias:

```
1  class AnimalTerrestre:  public Animal{
```

```
1  class AnimalTerrestre: virtual public Animal{
```

```
1  class AnimalAcuatico: public Animal{
```

```
1  class AnimalAcuatico: virtual public Animal{
```

Herencia múltiple

- Con la virtualización de las clases, es posible hablar de cierto parámetro o atributo que sea de una clase superior sin necesidad de hablar de la resolución de ámbito.

```
ombre: " + nombre + ". Especie: " + especie + ". Raza: " + raza + ". Color
e: " + AnimalAcuatico::nombre + ". Especie: " + AnimalTerrestre::especie +
```

- También es posible simplificar las construcciones ya que no es obligatorio crear una construcción en cadena.

```
tring raza, string color_de_pico) : AnimalTerrestre(nombre, especie), AnimalAcuatico(nombre, especie){
tring raza, string color_de_pico) : Animal(nombre, especie){
```

Copia de Objetos definición en la Construcción

- Existen múltiples formas de inicializar un objeto. Normalmente haciendo llamadas al constructor.

```
Silla A = Silla(); \\ A se guarda en una direccion del stack
                   \\ Silla() instancia un objeto A de la clase.
```

- Por otro lado, podemos crear un objeto a partir de otro:
  Silla B = A;
- Esta instrucción construye y almacena en memoria un objeto B con datos iniciales iguales a los que posea A.
- Se define esto como una construcción por copia, y, como toda construcción, está siempre definida implícitamente en todas las definiciones de clases.
- La firma de este constructor es Clase(Clase &otro) y, por supuesto, puede ser redefinida o sobreescrita en la implementación según la necesidad.

Copia de Objetos definición en la Construcción

- Para modificar un Constructor de Copia, habrá que sobreescribir la función Clase(const Clase& otro), en nuestro caso sería Pato(const Pato& otro);
- Es fácil notar que este constructor recibe una **referencia** a un objeto de la clase en cuestión que ya ha sido creado previamente, por lo que se supone que el Constructor de Copia realmente *copia* los datos de otro objeto ya instanciado.
- Para utilizar en el programa se debe declarar y construir el objeto en la misma línea:
  Clase c1;
  Clase c2 = c1; \\ Ocurre una construcción por copia
- Es muy importante notar que la declaración de la variable y su construcción por copia ocurre en la misma línea.
- Habrá que reimplementar el constructor de copia cuando alguno de los miembros sea un puntero, para proteger las direcciones de memoria (Ej. Punteros como cuentas de banco)
- Para evitar errores en clases con punteros, siempre se deben inicializar los punteros al puntero nulo o **nullptr** que es una dirección nula.

Copia de Objetos definición en la Construcción

- La clase Pato tiene un constructor por copia que copia la raza, además de copiar el nombre y la especie, pero no el color del pico.
- Así el método público Pato(const Pato& otro) solamente asigna this->raza = otro.raza, pero no copia el atributo color_de_pico.

```
1  Pato::Pato(const Pato& otro) : Animal(otro.nombre, otro.especie){
2      this->raza = otro.raza;
3      this->color_de_pico = "Desconocido";
4  }
```

Copia de Objetos definición en la Asignación

- Existen situaciones en las que queremos copiar un objeto a otro, pero ambos ya son objetos declarados en memoria.
- En este caso, el objeto ya está creado por lo que la asignación no llama a la construcción del objeto, directamente.
- Si observamos, la operación que se realiza es la de Asignación (=), en donde asignamos un objeto a otro, ambos ya declarados.
- Para sobrescribir el método de asignación por copia ha de reescribirse la función:

```
Clase& operator=(const Clase& otro);
```

- Es decir que, en la implementación, se debe:
- -Hacer copia de los atributos.
- -Recordar que this es un puntero a objeto, y que se debe retornar la indirección de dicho puntero
- Con esto, se retornaría un objeto nuevo, pero al retornar una referencia (Clase &) estamos devolviendo el mismo objeto que ha utilizado el operador.

```
1  Clase a, b;
2  b = a;
```

```
1  Clase& Clase::operator=(const Clase& otro) {
2      // hacer copia de los parametros;
3      this->param1 = otro.param1;
4      // retornar el objeto completo
5      return *this;
6  }
```

Copia de Objetos definición en la Asignación

- La palabra reservada `operator` seguida del símbolo = sobreescribe cada operación que utiliza el operador =.
- Como es de esperar, cuando hacemos copias por asignación se desea trabajar por referencias con objetos ya creados para mantener la eficiencia de la memoria, por eso el uso del operador `&`
- Como el objeto ya ha sido creado, retornar *this implica retornar el valor que desreferencia this, es decir, el objeto mismo como un valor copiado.
- Se diferencia de la construcción ya que son operaciones diferentes:

```
// Construcción por copia
Pato pato2 = pato1;
```

```
// Asignación por copia
pato2 = pato1;
```

Los operadores new y delete y la memoria del montón

- Para asignar objetos(y tipos primitivos) en la memoria del montón ha de utilizarse la palabra reservada new cuando se asigna la variable, pero cuidado que la palabra reservada new retorna un puntero y no el objeto.
- Fichero * a = new Fichero;
- Siempre debe eliminarse el puntero después de utilizar el operador.
- Esto se hace con la palabra reservada delete;
- delete a;

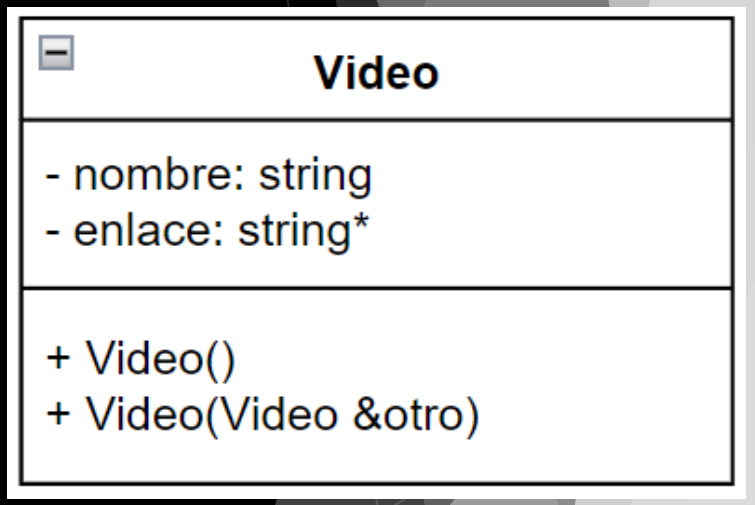
Los operadores new y delete y la memoria del montón

- El uso del operador new tiene ventajas:
  - Se permite asignar clases derivadas a clases principales: Animal *a = new Perro;
  - Permite almacenar memoria fuera del stack dejando más espacio libre para el programa principal.
- y desventajas:
  - La indirección es más lenta ya que el objeto está guardado muy lejos del stack.
  - Hay que eliminar el puntero new al finalizar el programa de forma manual: delete a;
- Su uso debe limitarse a cuando las ventajas sean necesarias.
- Debe tenerse cuidado a la hora de asignar datos ya que la instrucción new Clase inicializa con el constructor por defecto.
- Se permite inicializar con otros constructores usando new Clase (tipos …)

El operador new y el nullptr

- new y nullptr también se puede usar en punteros que sean miembros de un objeto:

```
Video::Video() {
    enlace = new string; // cuando debo crear un string nuevo en
el heap
    enlace = nullptr;   // cuando no debo crear un string en el
heap
}
```



- La diferencia está en que al usar new siempre es necesario eliminar el puntero atributo (¿en dónde?), en caso contrario, no se debe eliminar (si no se asigna como new en ningún momento).