Ingeniero en Software y Tecnologías Emergentes

Lenguajes de Programación Orientada a Objetos

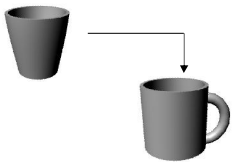
Programacion en C++
Herencia

ISyTE

1

Herencia

– Generación de nuevas clases en base a clases existentes

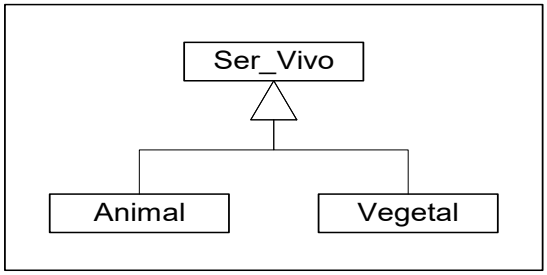


– La nueva clase (clase hijo, subclase o clase derivada) hereda las características de la clase base (clase padre, superclase o clase base)

2

Herencia

- Relación "*es un*"



Herencia

- En C++, una clase puede ser derivada de otra siguiendo la siguiente sintaxis.

```
class ClaseHijo: [public | protected | private] ClasePadre{
                    //Cuerpo de la clase
}
```

- La clase hijo heredará los miembros de la clase padre, pero solo tiene acceso directo a aquellos con nivel de protección *public y protected*
  - Los miembros *private* pueden ser accedidos por medio de funciones "setter" y "getter"

Herencia

- Cuando el modificador de herencia es public, los miembros de la clase base permanecen con el mismo nivel de protección en la clase derivada
- Cuando la herencia es protected, los miembros public y protected de la clase base se hacen protected en la clase derivada.
- Cuando la herencia es private, los miembros public y protected de la clase base se hacen private en la clase derivada.
- El tipo de herencia por defecto es private

5

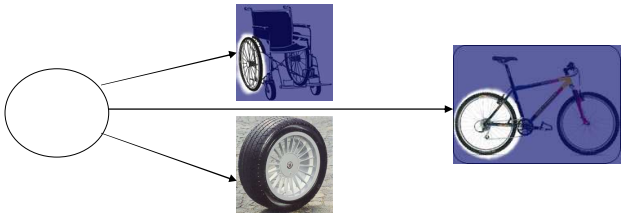
Herencia

```
class Persona{
  protected: int edad;
  public:
  char nombre[30];
  Persona(char elnombre[ ],int laedad){
        strcpy(nombre,el_nombre);
       edad=laedad;
  }
  saludar(){
             System.out.println("Como?");
             }
}
class Mexicano: public Persona{
}
```

6

Conceptos de POO

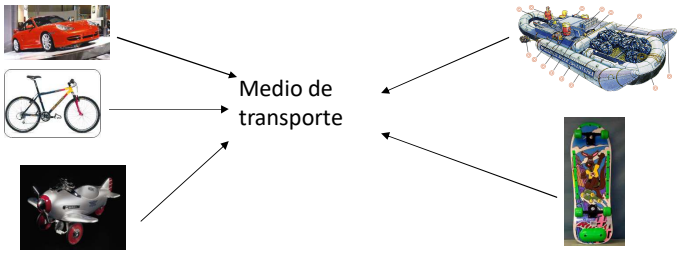
- Especialización:
  - *De lo general a lo particular*
  - *Las clases derivadas agregan funcionalidad*



7

Conceptos de POO

- Abstracción o generalización
  - *De lo particular a lo general*
  - *Una clase abtsracta agrupa varias clases con semántica común*



8

Herencia: Reimplementación o Sobreescritura

- Las funciones miembro en una clase base, pueden ser declaradas y reimplementadas en la clase derivada
- A esta acción se le llama “Sobreescritura” (*override* en ingles) o “reimplementación”
- Especializa funcionalidad ya definida en una clase derivada

Herencia: Reimplementación o Sobreescritura

```
class Persona{
  protected: int edad;
  public: char nombre[30];
  Persona(char elnombre[],int laedad){
          strcpy(nombre,el_nombre);
         edad=laedad;
  }
 void Saludar(){
              cout<<“Hola!”;
              }
}

class Profesionista: public Persona{
 public:
void Saludar(){
              cout<<“Soy profesionista!”;
 }
}
```

Herencia: Resolución de ámbito

- ¿Que pasa si en una clase derivada se quiere usar un método métodos de la clase padre que ha sido sobreescrito?
  - Se utiliza el operador de resolución de ámbito ::

```
class Profesionista: public Persona{
 public:
void Saludar(){
        Persona::Saludar();  //Invocando Saludar de Persona
         cout<<“Hola!”;
 }
}
```


Herencia: Reutilización de constructores

- Si en una clase padre existe un constructor default (sin parámetros), al construirse un objeto de clase hijo automáticamente se ejecuta el código del constructor default en la clase padre, seguido del código del constructor en la clase hijo
  - Se reutiliza automáticamente el constructor default.
- Si además del constructor default, la clase padre tiene constructores con parámetros, estos se pueden reutilizar al implementar el constructor.

Herencia: Reutilización de constructores

- Una constructor de ClaseX puede ser invocado automáticamente antes que el constructor de ClaseY, de la siguiente forma

```
ClaseY::ClaseY() : ClaseX(){
   //implementación de funcionY
}
```

- Esto se hace solo al implementar, no al declarar.

Herencia: Reutilización de constructores

- Se pueden enviar parametros al invocar automáticamente constructores:

```
class Profesionista: public Persona{
 Profesionista(char *elnombre,int laedad):Persona(elnombre,laedad)
  { };
 public void Saludar(){
          cout<<“Hola!”;
 };
}
```

Herencia Multiple

- Aunque no es recomendable, se puede derivar una clase a partir de 2 o más clases

```
class ClaseHijo: [public | private] ClasePadre1 [, [public | private] ClasePadre2]
    {
            //Cuerpo de la clase
    }
```

15

Herencia Multiple

```
class Musico{
    public void tocar();
}

class Cantante{
   public void cantar();
}
```

16

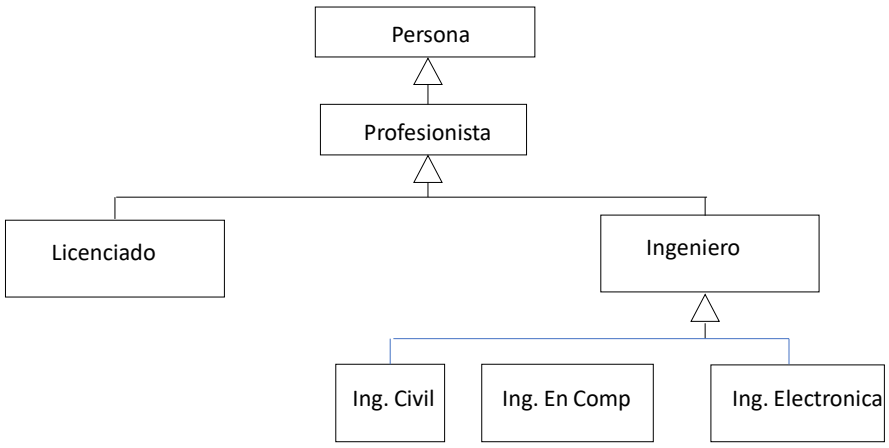
Herencia Multiple

```
class Mexicano: public Persona, public Cantante{
        public void saludar(){
                System.out.println("Hola");
          }
        public void cantar(){
                cout<<"De la sierra, morena…";
        }
}
```

- El mexicano es una persona y es cantante

Herencia Multiple

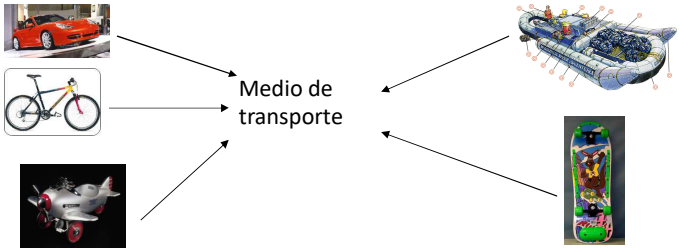
Implementar las clases para el siguiente diagrama de clases



Conceptos de POO

Abstracción o generalización

- *De lo particular a lo general*
- *Una clase abtsracta agrupa varias clases con semántica común*



Abstracción de clases

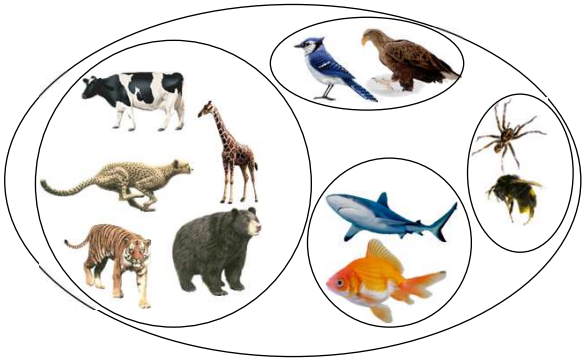
Si contamos con un conjunto de clases, estas clases pueden ser generalizadas en una clase abstracta

- La clase abstracta contendrá los elementos comunes en las demás clases
  - Si tenemos clases para diversos medios de transporte, estas pueden ser generalizadas como *MedioDeTransporte*
  - La clase abstracta debe tener lo que sea común para todos los medios de transporte (ejm. Avanzar, numero de pasajeros)

Polimorfismo

Polimorfismo

- *Dentro de un conjunto de entidades existen formas diferentes*



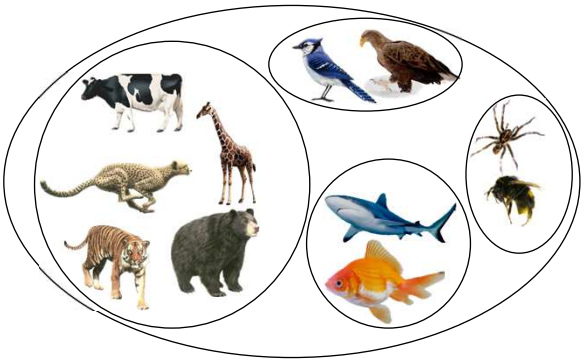
Animales

21

Conceptos de POO

• Polimorfismo

- *Cada diferente forma hace las cosas de forma diferente a las otras.*



Los animales
- Comen
- Duermen
- Hacen ruido

Pero lo hacen de forma diferente

Animales

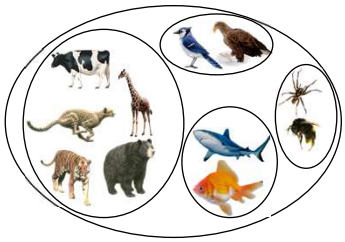
22

Conceptos de POO

- Polimorfismo

• *Se aprovecha gracias al principio de sustitución de Liskov:*

"Debe ser posible utilizar cualquier objeto instancia de una subclase en el lugar de cualquier objeto instancia de su superclase sin que la semántica del programa escrito en los términos de la superclase se vea afectado."



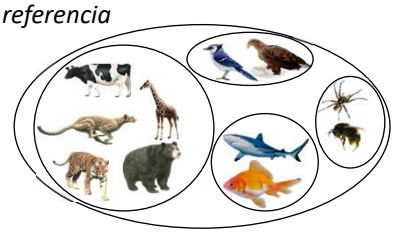
Animales

- Se puede escribir un programa que maneje animales
- Entonces podra manejar cualquier entidad del conjunto de animales
- No es necesario hacer un programa para Tigres, otro para Aguilas

Conceptos de POO

- El polimorfismo permite desencadenar operaciones distintas en base al mismo mensaje

- *Podemos referirnos a cualquier animal como un objeto de tipo animal.*
- *Si se envia un mensaje para que el animal coma, la operación de comer se ejecutará dependiendo del tipo de animal*
- *Debe comer como el animal de la instancia correspondiente, no de la referencia*



Animales

Abstracción de Objetos (Polimorfismo)

- Cualquier objeto de una clase derivada de otra puede ser referenciado de forma abstracta
- Se puede referenciar un objeto de una clase particular, como un objeto de una clase padre
  - Un Profesionista puede ser visto como Persona
  - Un Ingeniero puede ser visto como Profesionista y también como Persona

Abstracción de objetos (Polimorfismo)

- La referenciación abstracta o abstracción de objetos en C++ se hace por medio de apuntadores y referencias
  - Un apuntador de tipo Persona puede ser usado para manejar un objeto de tipo Ingeniero o un objeto de tipo Ing.Civil
  - Si el objeto fue creado de forma estática, siempre se puede acceder a su dirección de memoria con el operador de dirección &

Abstracción

```
Persona *lapersona, *lapersona2, *lapersona3,*lapersona4;
Profesionista *trabajador;
IngCivil el_civil("Jose Garza",45);
IngComputacion = new IngComputacion("Mark Gates",28);
lapersona2 = new IngElectronica("John Smith", 30);
trabajador = new Ingeniero("Juan Esparza",25);
lapersona3=&el_civil; //el civil es persona
lapersona4=trabajador; //trabajador es persona
```

Abstracción de objetos (polimorfismo)

- Cuando se utiliza la abstracción, solo se tiene acceso a los miembros de la clase abstracta (padre)
  - Si un Profesionista tiene Trabajar() y se deriva una clase Ingeniero de Profesionista, esta hereda Trabajar().
  - El Ingeniero adicionalmente puede tener un miembro Programar()
  - Si a un Ingeniero lo vemos como profesionista, no tendremos acceso a miembro Programar()

Polimorfismo

- Según el principio de sustitución de Liskow, se puede manejar cualquier objeto con referencias de clases padre, y al invocar una función sobrescrita se debe ejecutar la versión que sobrescribe, no la de la referencia utilizado

•Se puede referenciar a un *Ingeniero* como *Profesionista* y como *Persona*

•Si se invoca a un método de Persona que ha sido sobreescrito, se ejecutará la reimplementación, no la implementación el la clase referencia

29

Polimorfismo

```
Persona *lapersona, *lapersona2, *lapersona3,*lapersona4;
Profesionista *trabajador;
IngCivil el_civil("Jose Garza",45);
IngComputacion = new IngComputacion("Mark Gates",28);
lapersona2 = new IngElectronica("John Smith", 30);
trabajador = new Ingeniero("Juan Esparza",25);
lapersona3=&el_civil; //el civil es persona
lapersona4=trabajador; //trabajador es persona
trabajador->saludar();
lapersona->saludar();
Lapersona2->saludar();
Lapersona3->saludar();
//Todos saludan según el tipo de referencia que se utiliza
```

30

Polimorfismo

- En C++ es necesario utilizar funciones virtuales para hacer valido el principio de sustitución de Liskow.

```
class Persona{
  protected int edad;
  public char nombre[30];
  public Persona(char elnombre[],int laedad){
        strcpy(nombre,el_nombre);
       edad=laedad;
  }

public virtual void Saludar(){     //←- Funcion virtual
            cout<<“Como?”;
            }
}
```


Polimorfismo

- Cuando tenemos un objeto referenciado por un apuntador a una clase padre del objeto, si se invoca a una función virtual, se aplica el principio de sustitución

```
Persona *lapersona
lapersona = new Profesionista(“Juan Garza”,28);
Lapersona->saludar();
```

•En lugar de ejecutarse Persona::saludar(), se ejecuta Profesionista::Saludar()
•Se ejecuta la función correspondiente a la instancia, no a la referencia

Polimorfismo y herencia

- Ventajas
  - *Reutilización de código*
  - *Abstracción*
    - *No es necesario hacer código especializado para cada caso*
    - *No son necesarias variables especiales para cada objeto (solo es necesario saber la superclase)*

Funciones virtuales puras

- Al declarar una función miembro en una clase, esta puede ser declarada como virtual pura, es decir, sin código.
  - virtual void funcion()=0;
- Las funciones virtuales puras son utilizadas para establecer polimorfismo.
  - Podemos hacer una clase que servira como base para otras clases especializadas, especificando que comportamiento deben tener, pero sin decir como lo van a hacer (sin implementarlo)

Funciones y clases abstractas

- Por ejemplo, podemos considerar una clase Animal, que sirve para generalizar a todos los animales, y definir que atributos y comportamiento tienen en comun
  - Todos los animales tienen peso y tamaño.
  - Todos los animales comen, duermen y hacen ruido.
- La clase Animal servira como base para otras clases de animal especifico.

Funciones y clases abstractas

- Al referirnos a un Animal, sabemos que puede hacer ruido y comer, pero no sabemos como.
- Es necesario hacer estas funciones abstractas en la clase Animal
- El codigo especifico para decir como hace ruido un Perro, se implementa en la clase Perro (sobrescribiendo la funcion HacerRuido())
  - Se puede invocar Ladrar() en la funcion que sobrescribe.

Funciones y clases abstractas

```
class Animal{
   protected:
      int peso;
      int tamanio;
   public:
      Animal();
      virtual void HacerRuido()=0;
      virtual void Comer()=0;
      virtual void Dormir()=0;
}
```


Funciones y clases abstractas

- Cuando una clase tiene al menos una function virtual pura, automaticamente la clase se considera tambien abstracta.
- Una clase abstracta no puede ser instanciada.
  - Que codigo se ejecutaria si se invoca Comer() en una instancia de Animal?
- Es necesario derivar la clase para implementar el comportamiento abstracto.

Funciones y clases abstractas

```
class Perro:public Animal{
    public:
      Perro();
      void Ladrar();
      void HacerRuido();
      void Comer();
       void Dormir();
}
```

```
void Perro::Ladrar(){
     cout<<“barf!”<<endl;
}
void Perro::HacerRuido(){
    Ladrar();
}
void Perro::Comer(){
   cout<<“Comiendo un hueso”<<endl;
}
void Perro::Dormir(){
  cout<<“zzzzz….”<<endl;
```

Funciones y clases abstractas

- A un apuntador de clase abstracta podemos asignarle cualquier instancia de clases derivadas

```
int main(){
    Animal *animal=new Perro;

    animal->HacerRuido();
    //si vemos a un perro como animal, sabemos que puede hacer ruido
    //pero no sabemos que puede ladrar.
}
```

Tipo de objetos en apuntadores o referencias

- Con un apuntador de clase padre podemos referirnos a objetos de clases derivadas.
  - Cualquier objeto de clase derivada de Animal, puedo manejarlo con apuntador o referencia de Animal
- Al hacer esto, perdemos conocer detalles de las clases especificas
  - Si manejo un Perro con un apuntador de Animal, puedo saber las cosas que tiene como animal, pero no como Perro

Tipo de objetos en apuntadores o referencias

- En ocasiones es necesario convertir a un apuntador de clase especifica, para volver a tener esos detalles
  - Si recibo un animal, y si es Perro, hacer que este haga cosas especificas de Perro, como ladrar.
- Se puede conocer el tipo de una variable u objeto con el operador typeid.
- typeid devuelve un objeto de tipo type_info, el cual contiene una function name(), que devuelve el nombre de tipo del parametro enviado.

```
void FuncionDeAnimales(Animal *el_animal){
   //obtenemos el tipo de lo que contiene el apuntador el_animal
    type_info t=typeid(*el_animal);
     if(string(t.name())==“class Perro”){ //si el nombre del tipo es class Perro
        Perro *el_perro=(Perro *)el_animal; //convertimos el apuntador
         //para poder tener acceso a los elementos de Perro
         el_perro->Ladrar();
    }
}
```


Conversiones de tipo en objetos

- Cuando se hacen conversiones de tipo (casting) en apuntadores o referencias de clases en el mismo arbol de jerarquia de clases es conveniente usar el operador dynamic_cast.
- Este operador asegura que se conserve el polimorfismo apropiadamente y nos previene de errores.

- Prototipo:

dynamic_cast<tipo_destino>(variable_a_convertir)

- devuelve una variable de tipo tipo_destino

```
void FuncionDeAnimales(Animal *el_animal){
type_info t=typeid(*el_animal);
    if(t.name()==“class Perro”){

     //Convertimos con dynamic_cast
      Perro *el_perro=dynamic_cast<Perro *>(el_animal);

       el_perro->Ladrar();
    }
}
```


ISyTE

Sergio Omar Infante Prieto

sinfante@uabc.edu.mx