Tema 6: Herencia

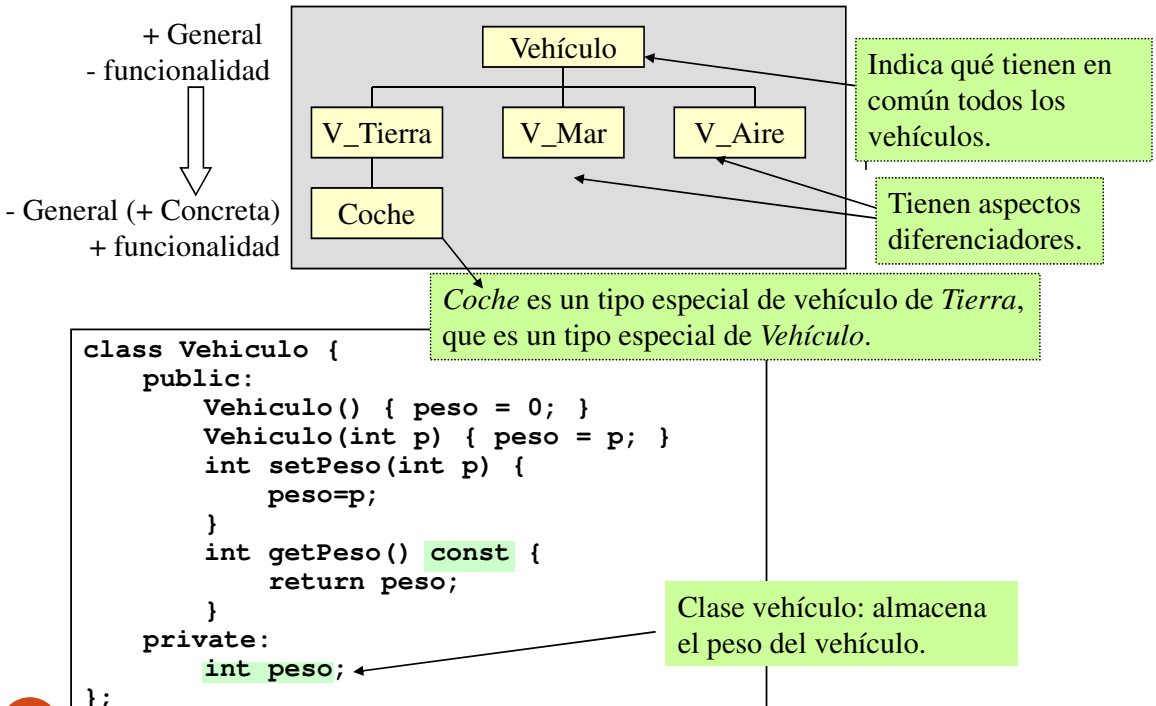
(El lenguaje C++)

Índice

1. Herencia

- En C++ pueden ser definidas clases en términos de otras clases (composición).
- En C++ también podemos definir una clase a partir de otra preexistente utilizando la **herencia** de clases.
- La herencia es una técnica de reutilización del software en la que se crean clases a partir de clases ya existentes por medio de la absorción de sus atributos y métodos, modificando aquellos que requiera la nueva clase o introduciendo nuevas capacidades.
- Al crear una nueva clase, el programador puede indicar que esta clase va a "heredar" los atributos y métodos de una clase ya existente que se llama **clase base**. La nueva clase se conoce como **clase derivada**.
  - La clase derivada tiene todas las funcionalidades de la clase base (**hereda** sus atributos y métodos).
  - Aparte, la clase derivada introduce nuevas funcionalidades (nuevos atributos y métodos).
  - La clase base es más general y la clase derivada más específica.

1. Herencia




```
class Vehiculo {
    public:
        Vehiculo() { peso = 0; }
        Vehiculo(int p) { peso = p; }
        int setPeso(int p) {
            peso=p;
        }
        int getPeso() const {
            return peso;
        }
    private:
        int peso;
};
```

1. Herencia

- Queremos representar un vehículo que viaje sobre tierra, para ello definimos una nueva clase: V_Tierra, que tendrá:

  Las funcionalidades de la clase Vehículo (peso)

  +

  Información adicional: velocidad del vehículo.

- El vehículo de tierra es un tipo especial de vehículo: la relación entre ellos más natural es la de la herencia.
- La clase *V_Tierra* **derivará** de la clase *Vehiculo* (clase base).
- Sintaxis para la derivación: **`class B: public A`** (la clase B deriva de A)

```
class V_Tierra : public Vehiculo
{
    public:
      V_Tierra(); //Constructor.
      V_Tierra(int p, float v);
      //interface
      void setVelocidad (float v) { velocidad = v; }
      float getVelocidad() const { return velocidad; }
      float MomentoInercia() { return getPeso()*velocidad; }
    private:
      float velocidad;
};
```

La clase *V_Tierra* contiene: las funcionalidades de la clase base *Vehiculo* **más** nuevas funcionalidades

Método público de Vehiculo

1. Herencia

- Ejemplo de utilización de la clase derivada:

```
int main()
{
  V_Tierra vehi(1200,145.3);
     cout << "Peso del vehículo" << vehi.getPeso()
         << "Velocidad del vehiculo"<< vehi.getVelocidad();
     return(0);
}
```

- Vemos dos características de la derivación:
  - ✓ Los miembros públicos de la clase base (ej. *getPeso()*) son heredados como públicos dentro de la clase derivada.
    - ✓ La función *getPeso()* no es un miembro directo de la clase *V_Tierra*. Esta función es un miembro implícito de esta clase, heredado de su clase padre *Vehiculo*.
  - ✓ Los miembros privados de la clase base (ej. *peso*) son heredados como privados dentro de la clase (*V_Tierr*). **NO** son accesibles por las funciones definidas fuera de la clase base, **NI SIQUIERA** por las funciones de la clase derivada.
    - ✓ Las funciones miembro de la clase derivada (ej. *MomentoInercia()*) sólo pueden acceder a ellos a través del interfaz público (*getPeso()*).

1. Herencia

- Una clase derivada (ej. *V_Tierra*) puede ser a su vez la clase base de otras nuevas clases:

Derivación anidada: Coche deriva de V_Tierra que deriva de Vehiculo.

```
class Coche : public V_Tierra
{
  public:
    //constructores
    Coche();
    Coche(int p, int  v, const string &n);
    //constructor de copia
    Coche ( const Coche & otro);

     //interfaz
    string getNombre() const;
    void setNombre(const string &n);
  private:
    string nombre;
};
```


Índice

2. Miembros protegidos

- Los miembros *públicos* de la clase base son accesibles desde cualquier función del programa, incluidas las funciones miembro de la clase derivada.
- Los miembros *privados* sólo son accesibles por las funciones miembro de la clase base, y por los amigos de la clase base. Estos miembros no son accesibles por las funciones de la clase derivada.
- En C++ existe además de éstos un nivel intermedio de proteccion entre el acceso público y el privado: **protected**.
  - Los miembros clasificados como ***protected*** en la clase base son accesibles por los miembros de las clases derivadas y por sus amigos.
  - Se comportan como ***public*** para los métodos de la clase derivada y amigas, y como ***private*** para el resto de funciones.
- Desde la clase derivada podemos hacer referencia a miembros públicos y protegidos de la clase base simplemente utilizando los nombres de éstos.

2. Miembros protegidos

- Si resulta imprescindible el acceso directo a los miembros privados de la base dentro de los métodos de la clase derivada, éstos deberán ser reclasificados como protegidos:

```
class Vehiculo {
   .........
   protected:
      int peso;
};

class V_Tierra : public Vehiculo
{
   .......
      float MomentoInercia() const {
       return peso * velocidad;
      }
   .......
   private:
      float velocidad;
};
```

Atributo protegido de Vehiculo

Atributo privado de V_Tierra

Índice

3. El constructor de la clase derivada

- Veamos los efectos de la herencia en el constructor de la clase deriv.
- Un objeto de una clase derivada contiene todos los atributos de la clase base.
  - Para inicializarlos de forma correcta **el constructor de la clase derivada invoca al constructor de la clase base**.
    - **Automáticamente**, sin necesidad de indicarlo explícitamente en el código: Se invoca al constructor por defecto de la clase base
    - Indicándolo **de forma explícita** en el código: Constructor con argumentos de la clase base.
- Al crear un objeto de la clase derivada, se ejecuta primero el constructor de la clase base y luego el de la clase derivada.

3. El constructor de la clase derivada

- Automáticamente

  El constructor de *V_Tierra* se podría implementar así:

```
V_Tierra :: V_Tierra( int p, int v)
{
   setPeso(p);
   setVelocidad (v);
}
```

LLama automáticamente al constructor por defecto de la clase *Vehiculo*

- El compilador de C++ genera el código para llamar al constructor por defecto de la clase base cada vez que es llamado un constructor de una derivada. Es decir:
  - Primero se llama al constructor por defecto de la clase *Vehículo*, que probablemente inicializará la variable *peso*.
  - Después se cambia el valor de *peso* llamando a la función *setPeso*.
- De forma explicita (una mejor solución para el ej anterior)

```
V_Tierra :: V_Tierra (int p, int v)
            : Vehiculo(p)
{
   setVelocidad(v);
}
```

LLama explicitamente al constructor de la clase *Vehiculo* con argumento.

Índice

4. Redefinición de funciones miembros

- Las acciones de las funciones definidas en la clase base pueden ser redefinidas en las clases derivadas.
- Ejemplo: vamos a crear una clase derivada de 'Coche' llamada 'Camion', en la que almacenaremos también el peso del trailer.

```
class Camion:public Coche
{
   public:
      Camion();
      Camion(int peso, int vel, const string &n, int peso_trailer);
      void setPeso(int peso, int peso_trailer);
      int getPeso() const;
   protected:
      int peso_remolque;
}

int Camion:: getPeso() const
{
   int peso_total;
       peso_total= Coche::getPeso() + peso_remolque;
       return(peso_total);
}
```

Creamos un nuevo método *getPeso*, que actuará de forma distinta en esta clase.

Para llamar a *getPeso* de la clase Coche hay que hacer uso de "**::**"

Índice

5. Herencia múltiple

- Hemos visto que una clase puede derivar de otra clase base.
- C++ permite también la herencia múltiple: una clase deriva de varias clases base, y por tanto hereda las funcionalidades de todas ellas.
- Ej: clase Motor que almacena información del motor (potencia, etc.)

```
class Motor {
  public:
    Motor();
    Motor(const string &nr_serie, int pot,
          const string &tipo_combus);
    //Interfaz
    void setNumSerie(const string &num_serie);
    string getNumSerie() const;
    void setPotencia(int pot);
    int getPotencia() const;
    void setTipoCombus(const string &tipo);
    string getTipoCombus() const;

  private:
    string numSerie, tipoCombus;
    int potencia;
};
```


5. Herencia múltiple

- Para representar un automóvil que tenga las propiedades de la clase Coche, y las de la clase Motor podemos utilizar derivación múltiple.

```
class CocheMotor: public Coche, public Motor
{
  public:
    CocheMotor();
    Coche Motor( int p, int v, const string &nom, const
              string &serie, int pot, const string &combus);
};
CocheMotor::CocheMotor (int p, int v, const string &nom,
      const string &serie, int pot, const string &combus)
            : Coche(p, v, nom), Motor(serie, pot, combus)
{

}
```

Mismo orden en que ha sido derivada

- La palabra **public** delante de Coche y de Motor, es necesaria, porque la derivación en C++, por defecto es privada.
- La derivación múltiple para formar la clase CocheMotor no añade funcionalidad, únicamente combina dos clases pre-existentes. Esto será muy útil para desarrollar clases a partir de otras más pequeñas.
- El constructor que espera seis argumentos no contiene código, el único propósito de éste es llamar a los constructores de las clases base.

Índice

6. Conversión entre clases base y derivada

- Cuando utilizamos herencia al definir una clase, podemos decir que un objeto de la clase derivada sea al mismo tiempo un objeto de la clase base.
- Esto tiene algunas consecuencias:
- **Conversiones en la asignación de objetos:**
- Si definimos dos objetos, uno de clase base y otro de la derivada:

```
Vehiculo v(900); //Vehículo con peso 900
Coche a(1200,130,"Ford"); //peso=1200, vel=130 y marca Ford
```

Como Coche es a su vez un Vehiculo podríamos hacer la asignación:

`v=a;` // El objeto **v** guardará el valor 1200 en su atributo peso

También podríamos intentar:

`a=v;` // El objeto **a** recibiría el peso, pero no la velocidad ni el nombre.
// Esta asignación NO será aceptada por el compilador.

- **Regla**: cuando se asignan objetos relaciones, una asignación en la que sobran parámetros es legal, pero no es legal si hay parámetro en blanco a la izquierda.

6. Conversión entre clases base y derivada

- La conversión de un objeto de la clase base a uno de la clase derivada puede ser definida explícitamente si se necesita.

  Creamos en la clase Coche, una función asignación que acepte como parámetro un objeto Vehiculo:

```
const Coche & Coche::operator=(const Vehiculo &veh) {
   setPeso(veh.setPeso());
   ..............
}
```

- **Conversión en la asignación de punteros.**
- Cuando llamamos a una función utilizando un puntero a un objeto, el tipo de puntero determina qué funciones miembro están disponibles.

```
Camion c;
Vehiculo *vp;
   vp = &c;
   ....
   float a = vp->getPeso();
```

Esto es válido, pero al llamar a la función *getPeso*(), estaremos llamando a la función de la clase Vehiculo.

6. Conversión entre clases base y derivada

- **Problema**: En el anterior ejemplo, la función *getPeso*() invocada será la de Vehiculo y el valor devuelto no será el correcto.

- También podemos hacer una conversión explícita:

```
Camion cam;
Vehiculo *vp;
   vp = &cam;
   .....
Camion *cam2;
   cam2 = (Camion *) vp;
```

vp apunta a un objeto Camion

convierte una variable Vehiculo* en una Camion* . Esto funcionará correctamente si en vp realmente apunta a un objeto Camion