

---

# Herencia

- **Necesidad de la herencia**
  - La mente humana clasifica los conceptos de acuerdo a dos dimensiones: **pertenencia** y **variedad**. Se puede decir que el Ford Fiesta es un tipo de coche (variedad o, en inglés, una relación del tipo **is a**) y que una rueda es parte de un coche (pertenencia o una relación del tipo **has a**).
  - Con la **herencia** se consigue clasificar los tipos de datos (abstracciones) por **variedad**, acercando así un paso más la programación al modo de razonar humano.
  - La herencia permite **definir una clase modificando una o más clases** ya existentes. Estas modificaciones consisten habitualmente en **añadir nuevos miembros** (variables o funciones), a la clase que se está definiendo, aunque también se puede **redefinir** variables o funciones miembro ya existentes. Cuando la implementación de un miembro de una clase base es inadecuada para una clase derivada el miembro puede sobreponerse en dicha clase derivada mediante una implementación adecuada.
-

---

# Herencia

- La clase de la que se parte en este proceso recibe el nombre de **clase base**, y la nueva clase que se obtiene se denomina **clase derivada**. Ésta a su vez puede ser **clase base** en un nuevo proceso de derivación, iniciando de esta manera una **jerarquía de clases**. De ordinario las **clases base** suelen ser **más generales** que las **clases derivadas**. Esto es así porque a las clases derivadas se les suelen ir añadiendo características, en definitiva variables y funciones que diferencian concretan y particularizan.
  - Uno de los problemas que aparece con la **herencia** es el del control del **acceso a los datos**.  
¿Puede una función de una **clase derivada** acceder a los datos **privados** de su **clase base**?  
No. Pero podría ser muy conveniente que una **clase derivada** accediera a todos los datos de su **clase base**.
-

---

# Herencia

- Existe el tipo de dato **protected**. Este tipo de datos es **privado** para todas aquellas clases que no son derivadas, pero **público** para una **clase derivada** de la clase en la que se ha definido la variable como **protected**.
  - Todas las funciones del programa pueden acceder a los miembros **public** de una clase base. Las funciones miembro y los **friend** de la clase base son los únicos que pueden acceder a los miembros **private** de una clase base.
  - El acceso **protected** es un nivel intermedio entre el acceso **public** y **private**. Los miembros y los **friend** de la clase base, y los miembros y los **friend** de las clases derivadas son los únicos que pueden acceder a los miembros **protected** de una clase base. Los miembros de las clases derivadas pueden hacer referencia a los miembros **public** y **protected** de la clase base indicando simplemente los nombres de los miembros
-

---

# Herencia

- El proceso de herencia puede efectuarse de dos formas distintas: siendo la clase base **public** o **private** para la clase derivada. En el caso de que la clase base sea **public** para la clase derivada, ésta hereda los miembros **public** y **protected** de la clase base como miembros **public** y **protected**, respectivamente. Por el contrario, si la clase base es **private** para la clase derivada, ésta hereda todos los datos de la clase base como **private**.
- Para indicar que una **clase deriva de otra** es necesario indicarlo en la **definición de la clase derivada**, especificando el modo **-public** o **private-** en que deriva de su **clase base**:

```
class Clase_Derivada : public o private Clase_Base
```

---

---

# Herencia

- Hay algunos elementos de la clase base que **no pueden ser heredados**:
  - Constructores
  - Destructores
  - Funciones **friend**
  - Funciones y datos estáticos de la clase
  - Operador de asignación (=) sobrecargado

- **Uso de las funciones miembro**

Una clase derivada no puede acceder directamente a los miembros private de su clase base. Si una clase base pudiera acceder a los miembros private de la clase base, se violaría el encapsulamiento de la clase base. Ocultar miembros private ayuda en la depuración y modificación correcta de los sistemas

---

---

# Herencia

- **Sobreposición de los miembros de la clase base en una clase derivada.**

Una clase derivada puede sobreponer una función miembro de su clase base indicando una nueva versión de dicha función con la misma firma. Cuando esto ocurre, lo más frecuente es que la versión de la clase derivada llame a la versión de la clase base a fin de que realice parte de la nueva tarea.

No utilizar el operador de resolución de ámbito (::) para referenciar la función miembro de la clase base hace que la función miembro de la clase derivada se llame a sí misma, provocando esto una recursión infinita.

(Ver ejemplo imprimir)

---

---

# Herencia

- **Uso de los constructores en las clases derivadas**

Debido a que una clase derivada hereda sus miembros de la clase base, cuando el objeto de la clase derivada es instanciado hay que invocar al constructor de la clase base para que inicialice los miembros de la clase base del objeto de la clase derivada (Se utiliza la sintaxis de inicializador de miembros).

Las clases derivadas no heredan los constructores y los operadores de asignación de la clase base. Sin embargo, los constructores y los operadores de asignación de la clase derivada llaman a los constructores y operadores de asignación de la clase base.

Si se omite el constructor en la clase derivada, el constructor de oficio de la clase derivada llama al constructor de oficio de la clase base

---

---

# Herencia

- **Conversión implícita de los objetos de la clase derivada**
  - A pesar de que un objeto de la clase derivada también “es un” objeto de la clase base, el tipo de la clase base y el tipo de la clase derivada son distintos. Bajo la herencia **public**, los objetos de la clase derivada se pueden tratar como objetos de la clase base, ya que la clase derivada tiene miembros que corresponden a cada uno de los miembros de la clase base. La asignación en el otro sentido no está permitida.
  - Con la herencia **public**, siempre es válido asignarle a un puntero de la clase base un puntero de la clase derivada. El puntero de la clase base sólo ve la parte de la clase base del objeto de la clase derivada. El compilador realiza una conversión implícita del puntero de la clase al puntero de la clase derivada. La asignación inversa no se hace implícitamente.
-

---

# Herencia

- **Relaciones “usa un”y “conoce un”**

Aunque un objeto de la clase persona no es un automóvil ni contiene a un automóvil, un objeto persona usa un automóvil. Una función usa un objeto emitiendo una llamada de función a una función miembro de dicho objeto.

Un objeto puede ser consciente de otro objeto. Esta relación se tiene cuando un objeto contiene un puntero o referencia a otro objeto.

## **Clase de estudio: punto, círculo, cilindro**

Tomaremos la jerarquía: punto, círculo y cilindro. La clase Punto está definida como:

---

---

# Herencia

```
// Punto.h
#ifndef PUNTO_H
#define PUNTO_H
class Punto {
protected:
    int x;
    int y;
public:
    Punto (int a=0, b=0);
    //Constructor predeterminado
    void setPunto(int,int);
    //Establece las coordenadas
    int getX() const {return x;};
    //obtiene coordenada x
    int getY() const {return y;};
    //obtiene coordenada y
    friend ostream & operator <<
        (ostream &,const Punto &);
};
#endif // PUNTO_H
```

---

---

# Herencia

```
//punto.cpp. Funciones miembro de Punto
#include<iostream.h>
#include "Punto.h"

Punto :: Punto(int a, int b){
    setPunto(a,b);}

void Punto::setPunto(int a,int b){
    x=a;
    y=b;
}

ostream & operator <<
(ostream &out, const Punto &p){
    out << "(" << p.x << "," << p.y << ")" <<
    endl;
    return out;
}
```

---

---

# Herencia

- Los datos miembro de **Punto** son **protected**. Así, cuando derivemos de Punto la clase Circulo, las funciones miembros de ésta podrán referenciar directamente las coordenadas x e y del punto, en lugar de tener que utilizar funciones de acceso. Esto puede dar como resultado un mejor desempeño. Un demostración del uso de esta clase podría ser la siguiente:

```
#include <iostream.h>
#include "punto.h"
void main(){
    Punto p( 72, 115 ); //Instancia el punto p
    //Datos protegidos de Punto, inaccesibles en //main
    cout << "La coordenada de x es "
    << p.getX() << endl;
    cout << "La coordenada de y es "
    << p.getY() << endl;
}
```

---

---

# Herencia

- La definición de la clase Punto y sus funciones miembro se pueden reutilizar en la nueva clase Círculo, definida como sigue:

```
//Circulo.h
#if !defined(CIRCULO_H)
#define CIRCULO_H
#include<iostream.h>
#include "Punto.h"
class Circulo : public Punto{
protected:
    double radio;
public:
    Circulo (double r= 0.0, int x=0, int y=0);
    void setRadio(double);
    double getRadio() const;
    double area ();
    friend ostream & operator << (ostream &,
const Circulo &);
};
#endif // CIRCULO_H
```

---

---

# Herencia

```
#if !defined(CIRCULO_H)
#define CIRCULO_H
#include<iostream.h>
#include "Punto.h"
class Circulo : public Punto{
protected:
    double radio;
public:
    Circulo (double r= 0.0, int x=0, int y=0);
    void setRadio(double);
    double getRadio() const;
    double area ();
    friend ostream & operator << (ostream &,
const Circulo &);
};
#endif // CIRCULO_H
```

---

---

# Herencia

```
#include "Circulo.h"

//El constructor de Circulo llama al constructor de
//Punto mediante el inicializador de miembros y luego
//inicializa el radio

Circulo :: Circulo (double r, int a, int b):
    Punto (a,b){ setRadio(r);}

void Circulo :: setRadio(double r)
{ radio = (r >=0 ? r : 0); }
double Circulo :: getRadio() const{
return radio;}

double Circulo :: area (){
return 3.14159 * radio * radio;}

//Salida de Círculo en el formato centro=(x,y); radio=r
ostream & operator <<
(ostream & out, const Circulo &c){
out << "centro=" <<"(" << c.x << "," << c.y
<< "); " << "radio=" << c.radio << endl;
return out;}
```

---

---

# Herencia

- La clase **Círculo** hereda de la clase **Punto** con herencia **public**. Esto significa que la interfaz **public** de **Círculo** incluye las funciones miembro de **Punto**, así como las las funciones de `Circulo.cpp`.
- Para reutilizar la definición del operador `<<` de la clase `Punto` en la definición del operador `<<` de la clase `Circulo` se hubiera podido hacer la implementación:

```
ostream & operator <<
(ostream & out, const Circulo &c){
out << "centro=" << static_cast<Punto> ( c )
<< ");radio=" << c.radio << endl;
return out;}
```

- La función `operator <<` de `Círculo`, que es friend de la clase `Circulo`, es capaz de tener como salida la parte `Punto` de `Circulo` convirtiendo mediante `cast` la referencia `c` de `Circulo` a un `Punto`. El resultado de esto es la llamada de a `operator <<` de `Punto` y el envío de la salida de del `Punto`.
-

---

# Herencia

- Definimos por último la clase **Cilindro** como derivada, mediante herencia public, de la clase **Circulo**. Así la interfaz public de Cilindro incluye tanto las funciones miembro de Punto y de Circulo, así como las de Cilindro.cpp.

```
// Cilindro.h
#ifndef CILINDRO_H
#define CILINDRO_H
#include "circulo.h"
class Cilindro: public Circulo{
protected:
    double altura;
public:
    //Constructor predeterminado
    Cilindro( double h=0.0, double r=0.0, int x=0, int
y=0);
    void setAltura (double); //Establece la altura
    double daAltura (); //Devuelve la altura
    double area () ; //Calcula y devuelve el área
    double volumen (); //Calcula y devuelve el volumen
    friend ostream & operator << (ostream &, const
Cilindro &);}
#endif // CILINDRO_H
```

---

---

# Herencia

```
#include <iostream.h>
#include "cilindro.h"
//El Constructor de Cilindro llama al de Circulo
//!y no al de Punto!

Cilindro::Cilindro(double h, double r, int x, int y):
    Circulo(r, x, y){setAltura( h );}
//Establece la altura del cilindro
void Cilindro::setAltura(double h){
    altura=(h >=0 ? h :0);}
//Obtiene la altura del cilindro
double Cilindro::daAltura(){ return altura;}
//Calcula el área del cilindro
double Cilindro::area(){
    return 2*Circulo::area() + 2* 3.14159 * radio * altura;
}
//Calcula el volumen del cilindro
double Cilindro::volumen () {
    return Circulo::area() * altura; }
//Envia a la salida las dimensiones del cilindro
ostream & operator <<
(ostream & out , const Cilindro & c){
    out << static_cast<Circulo> (c) << "; Altura=" <<
    c.altura;
    return out;}

```

---

---

# Herencia

```
// Prueba la clase cilindro
#include <iostream.h>
#include "Punto.h"
#include "Circulo.h"
#include "cilindro.h"
void main(){
//Crea objeto cilindro
    Cilindro cil (5.7, 2.5, 12, 23);
//Utiliza las funciones get para desplegar el cilindro
    cout << "La coordenada x es " << cil.getX()
        << "\nLa coordenada y es " << cil.getY()
        << "\nEl radio es " << cil.getRadio()
        << "\nLa altura es " << cil.daAltura() << endl;
//Utiliza las funciones para cambiar los atributos del
//cilindro
    cil.setAltura(10);
    cil.setRadio(4.25);
    cil.setPunto(2,2);
    cout << "Los datos de cil son: \n"
        << cil << endl;
//Despliega el cilindro como punto
    Point & pRef =cil; //pRef "piensa" que es un punto
    cout << "\n Cilindro impreso como punto es: "
        << pRef << endl;
//Despliega el cilindro como círculo
    Circulo & circuloRef =cil;
//circuloRef "piensa" que es un circulo
    cout << "\n Cilindro impreso como circulo es: "
        << circuloRef << endl;
}
```

---