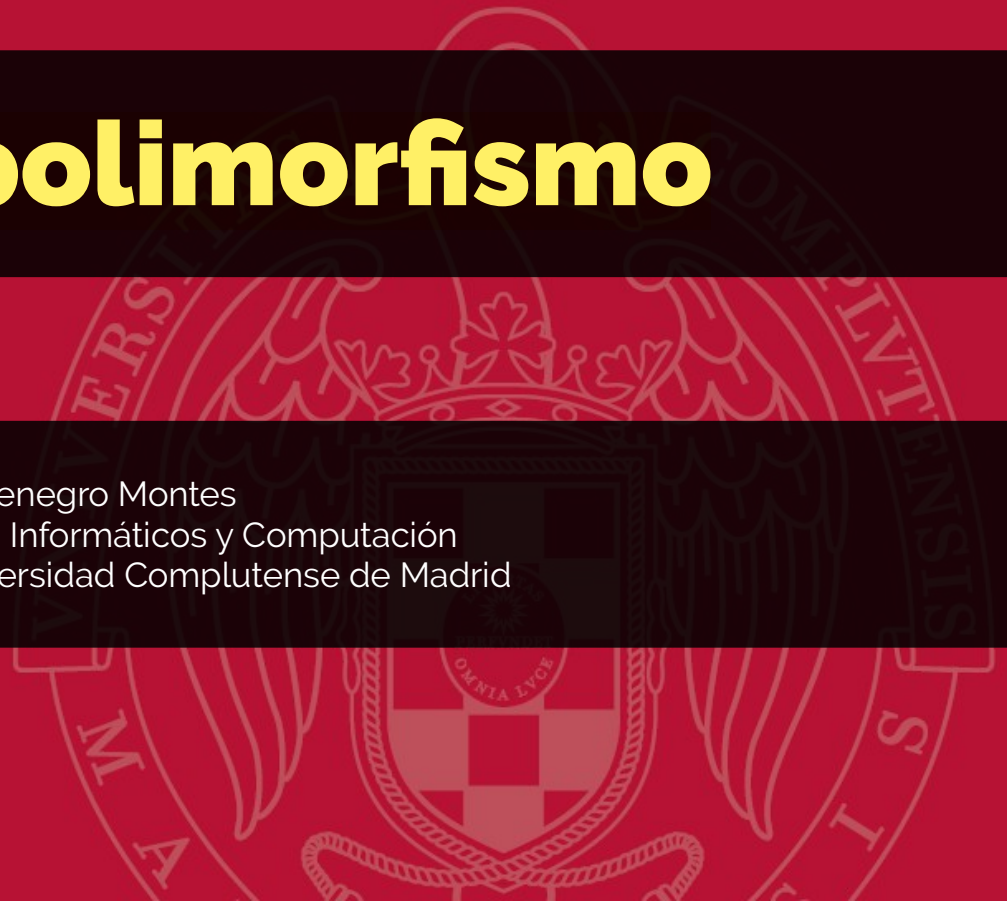


ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Herencia y polimorfismo

Manuel Montenegro Montes  
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid



En java se utiliza mucho. En la STL de c++ no se suelen utilizar. De hecho, el propio creador de la STL de C++ es muy crítico con la POO y con estas características propias de POO.

# Herencia

# Heredar de una clase

```
class Rectangulo {  
public:  
    Rectangulo(double ancho, double alto): ancho(ancho), alto(alto) { }  
  
    double area() { return ancho * alto; }  
    double perimetro() { return 2 * ancho + 2 * alto; }  
  
protected:  
    double ancho, alto;  
};  
  
class Cuadrado: public Rectangulo {  
public:  
    Cuadrado(double lado): Rectangulo(lado, lado) { }  
};
```

Cuadrado hereda de rectángulo.

Super

# Ejemplo

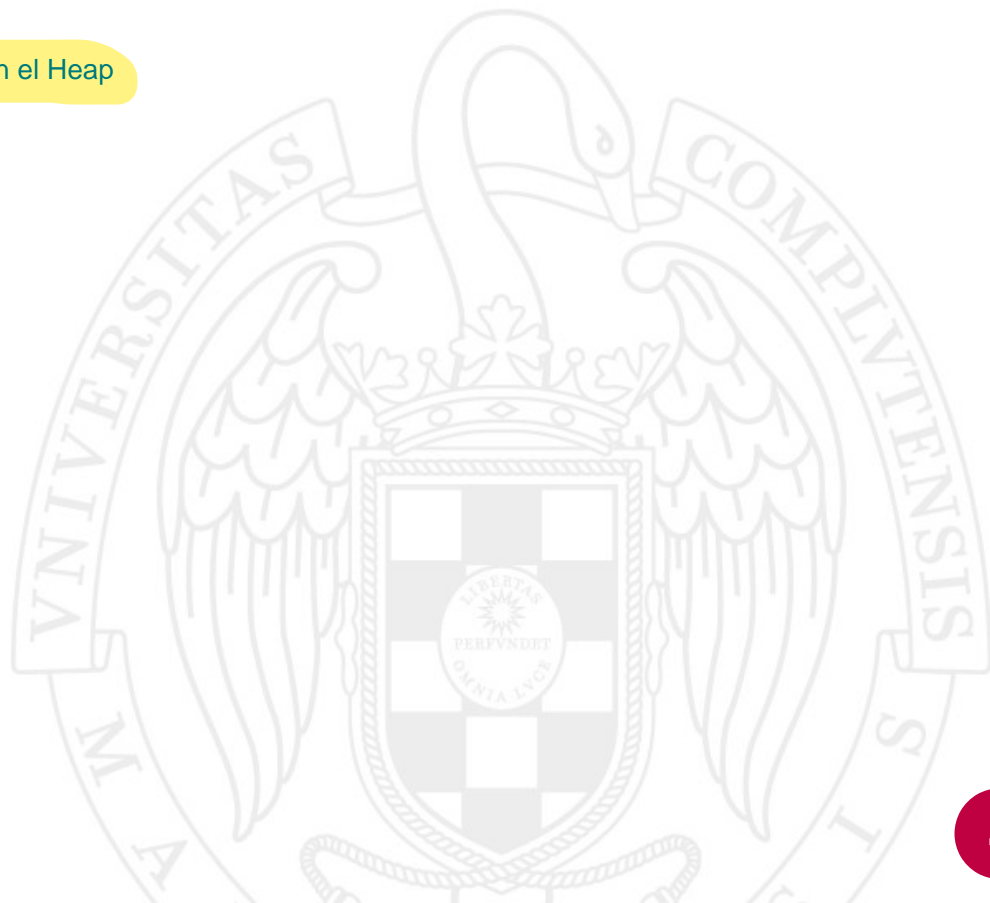
```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;
```

Solicitamos el ancho y el alto

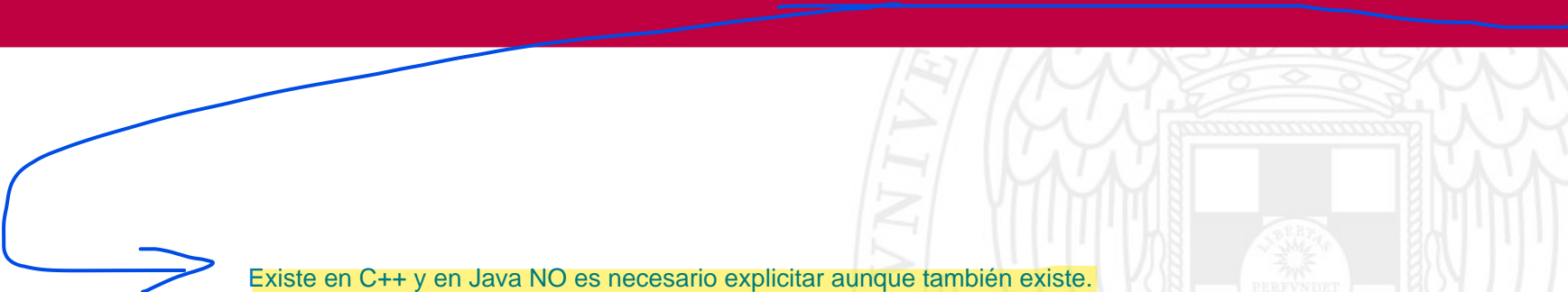
```
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}
```

Creamos los objetos en el Heap

```
double area = r->area();  
double perimetro = r->perimetro();  
cout << "Area: " << area << endl;  
cout << "Perímetro: " << perimetro << endl;  
  
delete r;
```



# Polimorfismo y métodos virtuales



Existe en C++ y en Java NO es necesario explicitar aunque también existe.

# Nuevo método: dibujar()

```
class Rectangulo {
public:
    ...
    void dibujar() {
        std::cout << "Rectángulo de ancho " << ancho << " y alto " << alto << std::endl;
    }
protected:
    double ancho, alto;
};

class Cuadrado: public Rectangulo {
public:
    Cuadrado(double lado): Rectangulo(lado, lado) { }

    void dibujar() {
        std::cout << "Cuadrado de lado " << ancho << std::endl;
    }
};
```

# Ejemplo

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
r->dibujar();  
delete r;
```

1.2 4.5

Rectángulo de ancho 1.2 y alto 4.5

1.2 1.2

Rectángulo de ancho 1.2 y alto 1.2



No nos imprime lo del objeto dibujar de la clase cuadrado.

# Vinculación estática vs dinámica

- C++ determina a qué método llamar en base al tipo del objeto sobre el que se realiza la llamada.

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}
```

`r→dibujar();`

```
delete r;
```

Método dibujar de la clase rectángulo.

`r` es de tipo puntero a `Rectangulo`  
Por tanto el compilador determina que  
`r→dibujar()` llama al método  
`dibujar` de `Rectangulo`.



# Vinculación estática vs dinámica

- Si se realiza **vinculación dinámica**, decimos al compilador que se compruebe, en tiempo de ejecución, la clase a la que pertenece el objeto, y se llame al método correspondiente a esa clase, independientemente del tipo.
- Por defecto, en C++ se utiliza vinculación estática.
- Por defecto, en Java se utiliza vinculación dinámica.

En Java NO existe la vinculación estática.

# Habilitar la vinculación dinámica

```
class Rectangulo {  
public:
```

CON ESTO COMPRUEBA LA CLASE DESDE LA QUE SE ESTÁ LLAMANDO AL MÉTODO EN TIEMPO DE EJECUCIÓN.

```
    virtual void dibujar() {  
        std::cout << "Rectángulo de ancho " << ancho << " y alto " << alto << std::endl;  
    }  
protected:  
    double ancho, alto;  
};
```

```
class Cuadrado: public Rectangulo {  
public:  
    Cuadrado(double lado): Rectangulo(lado, lado) { }  
  
    virtual void dibujar() {  
        std::cout << "Cuadrado de lado " << ancho << std::endl;  
    }  
};
```

No sería necesario, ya que ya lo tiene la superclase. Sin embargo, es recomendable.

# Ejemplo

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
r->dibujar();  
  
delete r;
```

1.2 4.5

Rectángulo de ancho 1.2 y alto 4.5

1.2 1.2

Cuadrado de lado 1.2



Ahora, al ser un método virtual ya se imprime de la manera correcta.

# Reglas generales

- Cualquier método que sea susceptible de ser reescrito debe declararse como `virtual`.
- Si una clase tiene un método `virtual`, es muy aconsejable declarar su destructor como `virtual`, aunque no haga nada.



# Métodos abstractos

```
class Figura {  
public:  
    virtual double area() = 0;  
    virtual double perimetro() = 0;  
    virtual void dibujar() = 0;  
  
    virtual ~Figura() { }  
};  
  
class Rectangulo: public Figura { ... }
```

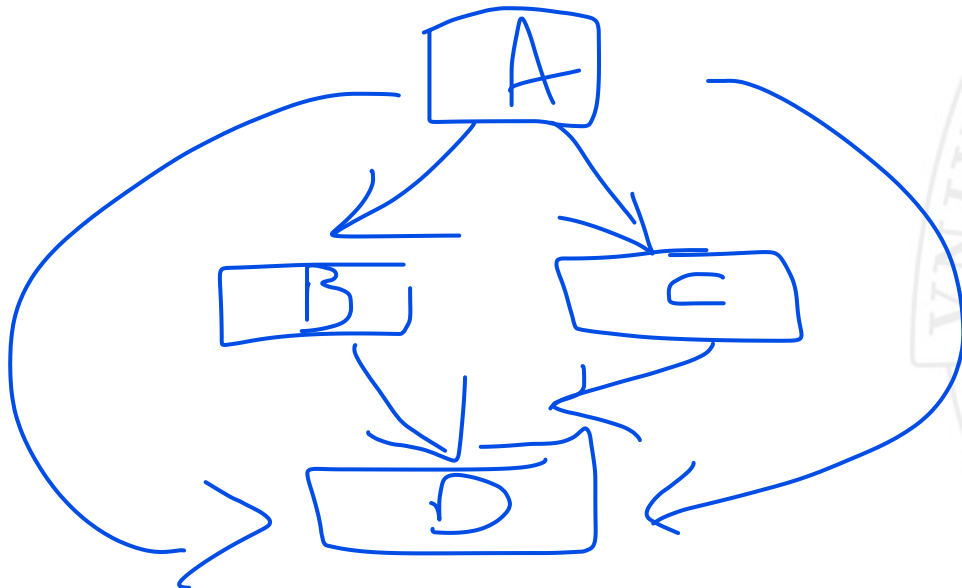
Igualar el método a 0 hace que los métodos sean abstractos.

- Los métodos abstractos han de ser virtuales. Tiene lógica, si es abstracto es porque queremos que sea reescrito en las hijas.
- Si una clase tiene un método abstracto, la clase es abstracta.
  - No pueden crearse instancias de **Figura**.

# Otras diferencias con Java

- En C++ no existe la noción de interfaz (interface).
- Se permite herencia múltiple.

EN JAVA NO



D heredaría de A los atributos de manera doble, por lo cual, estas herencias múltiples pueden llegar a dar lugar a errores.